The LogicGuard Stream Monitor Specification Language*

Tutorial and Reference Manual
(Version 1.01)

Wolfgang Schreiner, Temur Kutsia, David Cerna
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at, Temur.Kutsia@risc.jku.at
David.Cerna@risc.jku.at

Michael Krieger, Bashar Ahmad
RISC Software GmbH, Hagenberg, Austria
Michael.Krieger@risc-software.at, Bashar.Ahmad@risc-software.at

Helmut Otto, Martin Rummerstorfer, Thomas Gössl
SecureGUARD GmbH, Linz, Austria
hotto@secureguard.at, mrummerstorfer@secureguard.at
tgoessl@secureguard.at

November 16, 2015

Abstract

This report describes the design and use of the LogicGuard language for specifying stream monitors. These monitors observe streams of values (e.g., messages flowing through a network connection) and check whether the streams fulfill desired safety properties. These properties are described on a very high level of abstraction in a purely declarative way by notions that are derived from classical predicate logic, in particular by logic formulas that are quantified over stream positions. To raise the level of abstraction, auxiliary internal streams can be specified whose values are constructed from the values on the external streams by notions that are thar are similar to classical set builders. From the abstract specifications automatically executable monitors are generated which surveil the streams in real time and trigger warnings if violations of the specified properties are observed.

*Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the projects 832207 “LogicGuard” and 846003 “LogicGuard II”.

Contents

1 Introduction

The goal of the LogicGuard project [1] is the development of a language that allows

- to specify properties of (potentially infinite) streams of values on a very high level of abstraction, and
- to generate executable monitors that observe the streams at real time for violations of these properties.

Our target are streams of messages that flow through network connections and are to be monitored for the violation of security properties.

To achieve the high level of abstraction, the specification language is purely declarative, i.e., it describes properties, not programs (even not in the form of e.g. recursive function definitions, which also have an operational flavor). Furthermore, we rely on formalisms that are widely known and well understood, not on unfamiliar special-purpose concepts. More precisely, the LogicGuard specification language (of which earlier forms are described in [2, 3]) is based on the classical notions of *first-order predicate logic* and *set theory*. Logical formulas that are quantified over stream positions describe properties of streams by relating the values of messages at different stream positions with each other. These properties are not necessarily expressed in terms of the original low-level streams: one may apply stream builders that are similar to classical set builders to raise the level of abstraction by constructing new streams whose values e.g. comprise multiple values of the original stream. By these concepts complex properties can be specified in a very elegant way.

The current version of the LogicGuard system indeed implements (on top of the .NET framework and the programming languages F# and C#) the full language as described in this report (including the "history pruning" optimization that was described in [5] for an earlier version of the language). The current state of the system is always visible on the project's home page

`http://www.risc.jku.at/projects/LogicGuard2`

In a nutshell, our goals have been achieved: with the LogicGuard language and system we are able to monitor live network traffic for the violations of properties specified on a very high level of abstraction.

The remainder of this document is structured in two parts, a tutorial (Sections 2–4) and a reference manual (Appendices A–M). In Section 2 we give by a very simple example a quick introduction into the practical use of the language. In Section 3, we describe some principles underlying the design and implementation of the language as a basis for understanding the operational behavior of runtime monitors created from specifications. Section 4 demonstrates by more comprehensive examples some more advanced features of the language. In Section 5 we describe those aspects of the language and the implementation that still need further elaboration.

In Appendix A, we describe how to run the LogicGuard system. In Appendix B, the lexical syntax of the language is presented while in Appendix C its static type system is sketched. Appendices D–L describe the context free grammar of the various phrases of the language and explain their informal semantics and pragmatic use. Finally Appendix M documents the language's formal grammar in the notation of the ANTLR 4 parser generator [4] that was used to produce the lexical and syntactic analyzer of the system.

2 A Quick Start

For the impatient reader, we demonstrate by some simple examples what the specification language and monitoring system is all about.

First, let us assume that for a given stream *IP* of integers, e.g.,

$$2,3,5,-1,6,5,4,-1,2$$

we would like to check whether the stream *S* derived from *IP* by adding one to every value

$$3,4,6,0,7,6,5,0,3$$

does not contain any zeros (this property is in above example violated by the two stream positions 3 and 7).

In order to generate a corresponding monitor, we write the specification which is depicted in Figure 1 and contained in the plain text file `quick.lgs`.

This specification first declares an external type `int` (this is only used for type checking; system does not care the values of this type are), then a predicate `IsLogical`, and finally a function `Increment`. These functions are not defined in the specification but are provided by an external .NET library. They could e.g. be provided by a C# class

```
using System;
namespace External
{
  class IntFunctions
  {
    public static bool IsZero(Int64 m) { return m == 0; }
    public static Int64 Increment(Int64 m) { return m+1; }
    public static bool Print(Int64 m)
    { Console.WriteLine("Print: "  + m); return true; }
    ...
  }
}
```

that is compiled to a library file `External.dll`. For debugging purposes, the specification also declares the external predicate `Print` which always returns true but also prints its argument as a side effect.

The specification then declares the external integer stream `IP` which will be filled with values from an external source (the name `IP` is determined by the runtime engine, see below). It also defines the internal integer stream `S` that is constructed by reading in stream `IP` the value at every position `x` in `IP`, incrementing this value, and placing the result in `S`. The notation in the definition resembles the conventional set builder notation

$$S := \{x \in IP \ : \ Increment(x)\}$$

that builds from a set *IP* another set *S*.

```
// ------------------------------------------------------------------------
// quick.lgs
// a quick start into LogicGuard
//
// Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD).
// All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard/
// ------------------------------------------------------------------------

// an externally defined type
type int;

// an externally defined predicate
logical IsZero(value<int> x);

// an externally defined function
value<int> Increment(value<int> x);

// for debugging: an externally defined predicate
// that returns always "true" but prints its argument as a side effect
logical Print(value<int> x);

// we use the integer engine
stream<int> IP;

// stream S that adds one to the elements of IP
stream<int> S = stream<IP> x : Increment(@x);

// monitor that just prints the streams
monitor<S> PrintS  = monitor<S> x: Print(@x);

// a monitor that checks that there is no zero on S
monitor<S> M = monitor<S> x: !IsZero(@x);

// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

Figure 1: A Simple Specification

```
LogicGuard 1.01 (November 16, 2015)
Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD)
All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard2/
------------------------------------------------------------------------
Parsing specification <quick.lgs>.
Reading file <u:\projects\logicguard-2014\logicguard2\ap2\logicguard2\main\specs
\quick.lgs>.
No syntax errors.
No type errors.
No analyzer errors.
Monitoring started.
0: 2#103
Print: 3
1: 3#105
Print: 4
2: 5#107
Print: 6
3: -1#111
Print: 0
VIOLATION<M>: position<S> x=3
4: 6#113
Print: 7
5: 5#117
Print: 6
6: 4#123
Print: 5
7: -1#129
Print: 0
VIOLATION<M>: position<S> x=7
8: 2#130
Print: 3
Message trace is completed.
Press <ENTER> to continue.
```

Figure 2: A Monitor Execution

The specification declares a monitor `M` which, for every position `x` in `S`, checks whether the stream value at that position is not zero (and reports a violation, otherwise). The notation in the definition resembles the logical notation for universal quantification

$$M :\Leftrightarrow \forall x \in S : \neg IsZero(x)$$

which defines predicate M to be true if all elements in S have the desired property (the analogies between `S` and S and between `M` and M are simplified, because in the specification variable `x` actually denotes stream positions, not stream values; see Section 4 for closer analogies).

For debugging purposes, the program also declares a "pseudo-monitor" `PrintS` that prints the content of `S` and does not report a violation.

The stream to be checked may be contained in a file `quick.txt` with the following contents:

```
2  103
3  105
5  107
-1 111
6  113
5  117
4  123
-1 129
2  130
```

The first element of every line represents a value of the stream, the second element represents the time of that value (which is not used in our example).

We may then start the program from the command line as follows:

```
LogicGuard -engine int -verbose
  -input Main/traces/quick.txt quick.lgs External.dll
```

The command line option `-engine int` indicates the use of the "integer engine" that reads files in the format given above and generates a corresponding stream `IP` of integer values. The argument `quick.lgs` denotes the name of the specification file, the argument `quick.txt` the name of the stream file; the argument `External.dll` denotes the library where the definitions of the external functions and predicates can be found.

The output of the program is then as depicted in Figure 2. After having printed the startup message, the system reads the specificaton and parses, type-checks, and analyzes it. It then translates the specification into an executable monitor and starts the monitor feeding it the values from the stream file. During the execution of the monitor, each value in the external stream `IP` is printed in the following format:

⟨*position*⟩`:` ⟨*value*⟩ `#` ⟨*time*⟩

For every value in the external stream `IP`, the pseudo-monitor `PrintS` just prints the corresponding value in the internal stream `S`. The real monitor `M`, however, reports two violations for positions $x = 3$ and $x = 7$ in S. When the message trace is completed, the program terminates.

While above example illustrated some general principles of the monitoring language, we are now going to demonstrate the usage of the system for monitoring a live stream of network

```
// ------------------------------------------------------------------------
// network.lgs
// a minimal specification to test network traffic
//
// Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD).
// All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard/
// ------------------------------------------------------------------------

// an externally defined type
type datagram;

// we use the ip engine
stream<datagram> IP;

// a monitor that invokes a dummy predicate on every message of IP
monitor<IP> M1 = monitor<IP> x: true;

// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

Figure 3: A Simple Network Traffic Specification

```
LogicGuard 1.0 (October 26, 2015)
Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD)
All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard2/
------------------------------------------------------------------------
Parsing specification <network.lgs>.
Reading file <u:\projects\logicguard-2014\logicguard2\ap2\logicguard2\main\specs
\network.lgs>.
No syntax errors.
No type errors.
No analyzer errors.
Monitoring started.
Press any key to stop monitoring.
0: from 193.170.38.171:55263 to 65.52.0.51:5671 (37 bytes)#1428922916490
1: from 65.52.0.51:5671 to 193.170.38.171:55263 (0 bytes)#1428922917491
2: from 65.52.0.51:5671 to 193.170.38.171:55263 (37 bytes)#1428922917491
3: from 193.170.38.171:55263 to 65.52.0.51:5671 (0 bytes)#1428922917491
4: from 193.170.38.171:55493 to 193.170.37.138:80 (0 bytes)#1428922939494
5: from 193.170.37.138:80 to 193.170.38.171:55493 (0 bytes)#1428922939494
6: from 193.170.38.171:55493 to 193.170.37.138:80 (0 bytes)#1428922939494
7: from 193.170.38.171:55494 to 193.170.37.223:995 (0 bytes)#1428922940899
...
Message trace is completed.
Press <ENTER> to continue.
```

Figure 4: A Monitor Execution

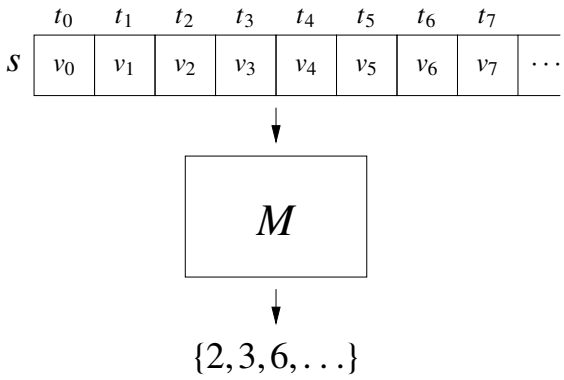

Figure 5: The Static View of a Monitor

traffic. For this purpose, we take the specification in Figure 3 that describes a minimal network monitor that processes every IP datagram by applying the dummy predicate `true` to it. While this monitor will thus certainly not detect any property violation, we can nevertheless start it as follows:

```
LogicGuard -engine ip -verbose
  -interface F0DEF1D6B70D network.lgs External.dll
```

The command line option `-engine ip` indicates that we would like to monitor IP datagrams while the option `-interface` denotes by six pairs of hexadecimal digits the address of the network interface from which we would like to capture these datagrams (on a machine running Microsoft Windows, executing `ipconfig /all` displays every available network interface and its address under the title "Physical Address"). By using the option `-verbose` every datagram observed on the interface is printed in the format depicted in Figure 4 indicating the source and destination IP address and the size of its payload. The monitoring stops when on the console any key is pressed.

By applying more complex specifications, we can thus monitor live network traffic for the conformance to specific securicy policies.

3 The Execution Model

Before we delve deeper into the details of the specification language, we are going to describe the problem that it addresses; without going into too much formal detail, we would like to convey an intuitive picture of how runtime monitoring is implemented as a basis for writing elegant and efficient stream monitor specifications.

Monitors: The Static View Our goal is to describe monitors that accept "streams", i.e., potentially infinite sequences of *messages*, and determine whether the stream violates some expected property. An example of such a stream may be the sequence of packets flowing through an Internet connection where the desired property is the absence of any attempt to connect from the Internet without appropriate authorization to some computer in the local network.

Such a monitor is not very useful, if it just terminates when it has detected a violation; rather it shall continue to investigate the stream for any more violations. Since the stream is potentially infinite, the monitor may thus run for an unbounded amount of time and report an unbounded number of violations. If we assume that each message in the stream can cause a violation of the property, we may uniquely identify each violation with a stream position.

If we also assume that a monitor guards a single stream (as we will see, a monitor may also guard multiple ones), then we can model the monitor by a function M from streams to sets of stream positions (which identify property violations):

$$M : \mathit{Monitor}$$
$$\mathit{Monitor} := \mathit{Stream} \to \mathbb{P}(\mathit{Position})$$

In other words, for a given stream s, $M(s)$ denotes the set of violating positions of the stream. This view is illustrated in Figure 5.

As already stated, a stream is a finite or infinite sequence of messages while a stream position is just a natural number:

$$\mathit{Stream} := \mathit{Message}^* \cup \mathit{Message}^\omega$$
$$\mathit{Position} := \mathbb{N}$$

A message (v,t) carries an unspecified *value* v but also has a *time* t associated which indicates when the message has arrived in the stream:

$$\mathit{Message} := \mathit{Value} \times \mathit{Time}$$
$$\mathit{Value} := \ldots$$
$$\mathit{Time} := \mathbb{N}$$

Times are identified by natural numbers, because because we assume that a time can be only measured by the discrete "ticks" of a digital clock. We assume that the messages (v_i,t_i) in a stream are ordered according to their times, i.e., $v_i \leq v_{i+1}$; the inequality is not strict, because two messages may arrive within the same clock tick.

Monitors: The Operational View While the static view of a monitor as a function from streams to sets of stream positions represents a nice mathematical model, it does not adequately capture the monitor's operational behavior: actually, M observes only one message at a time; for each message M decides (based on its current state resulting from the observations of the previous messages), whether it can report some violations or not and then changes its state to accommodate the new observation.

This view is illustrated in Figure 6. For every message received, the monitor reports a (potentially empty) set of violating positions. Please note that a violating position may be reported at any step, not only at the step corresponding to the position; furthermore, more than one position at a time may be reported. For example, while in the figure position 3 is indeed reported in step 3, position 2 is (together with position 6) reported in step 6.

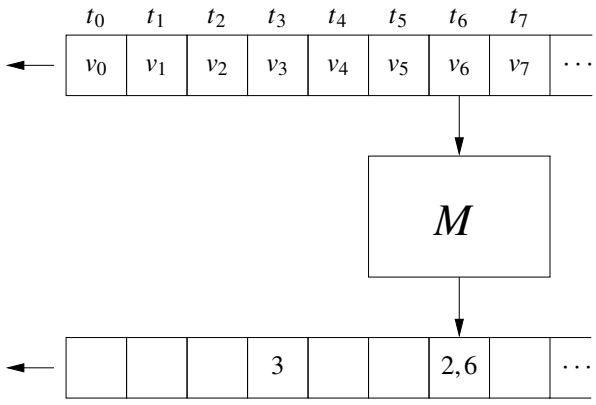

Figure 6: The Operational View of a Monitor

The corresponding mathematical model of the monitor is a function M from messages to answers which describe only part of the overall result:

$$M : MonitorStep$$
$$MonitorStep := Message \to MonitorAnswer$$
$$MonitorAnswer := \mathbb{P}(Position) \times (done \ + \ next \text{ of } MonitorStep)$$

For a given message m, $M(m)$ denotes a pair of a set of violating positions and an indication about the future of the monitor: it either just terminates (result *done*) or it switches to a new state M' in which it is ready to accept the next message (result $next(M')$).

For an initial state M_0 of the monitor and a stream $m_0, m_1, \ldots$ of messages, we thus retrieve by repeated monitor application

$$M_0(m_0) = (p_0, next(M_1))$$
$$M_1(m_1) = (p_1, next(M_2))$$
$$M_2(m_2) = (p_2, next(M_3))$$
$$\ldots$$

a sequence of sets of violating positions $p_0, p_1, p_2, \ldots$ whose union denotes the set of all violating positions reported by the monitor.

We can describe the operational behavior of the monitor more precisely by the following function:

$$run : MonitorStep \times Stream \times \mathbb{N} \to \mathbb{P}(Position)$$
$$run(M, s, n) :=$$
    if $n = 0$ then
        $\emptyset$
    else
        let $m := head(s)$
        match $M(m)$ with
            $\mid (rs, done) \to rs$
            $\mid (rs, next(M')) \to rs \cup run(M', tail(s), n - 1)$

Given a monitor M and stream s, $run(M, s, n)$ denotes the set of violating positions that M reports in n steps.

Monitors from Specifications Actually our goal is not to program a monitor M from scratch but to have it generated from an abstract specification S. The *meaning* $[\![S]\!]$ of S can be formally defined by a denotation function

$$[\![.]\!] : \mathit{Specification} \rightarrow \mathit{Monitor}$$

In other words, $[\![S]\!]$ is the static view of a monitor described above.

However, the *implementation* $T(S)$ of S can be only constructed by a translation function

$$T : \mathit{Specification} \rightarrow \mathit{MonitorStep}$$

In other words, $T(S)$ is the operational view of a monitor described above.

Denotation and translation are related by the following constraint:

$$\begin{aligned} &\forall S \in \mathit{Specification}, s \in \mathit{Stream}, n \in \mathbb{N} : \\ &\quad run(T(S), s, n) \subseteq [\![S]\!](s) \end{aligned}$$

In other words, any execution of $T(S)$ may only report positions that are considered as violating by $[\![S]\!]$. The converse is not necessarily true, i.e., $[\![S]\!]$ may consider a position as violating that is never reported by $T(s)$. The reason is that, while the denotation $[\![.]\!]$ has a "global" view on an infinite stream s, T only sees a finite prefix and not every property that is violated by s is necessarily also violated by any finite prefix of s. As a principle restriction, a monitor may only detect violations of so-called *safety properties* for which it is sufficient to consider finite prefixes.

The Components of a Specification We are now going to look in somewhat more detail into the structure of a specification. A specification is composed of various kinds of syntactic phrases, most important:

- *Monitors*: a single specification may describe multiple monitors.
- *Streams:* the monitors may observe multiple streams which may be
  - *external*: their messages come from external sources;
  - *internal*: their messages are internally constructed.
- *Formulas:* a formula describes a property that has to be satisfied by the messages of some stream(s).
- *Terms:* a term describes a value (object) that can be computed from the messages of some stream(s).

Like monitors, the meaning of all these phrases can be denotationally defined by a global view on the streams. In the implementation, however, each entity is translated to an operational

"engine" which is similar to a monitor as described above; the difference is only in the kind of objects that the engines return at every step and upon termination:

$$
\begin{aligned}
\mathit{MonitorStep} &:= \mathit{Present} \to \mathit{MonitorAnswer} \\
\mathit{MonitorAnswer} &:= \mathbb{P}(\mathit{Position}) \times (\mathit{done} \ + \ \mathit{next} \ \mathsf{of} \ \mathit{MonitorStep}) \\
\mathit{StreamStep} &:= \mathit{Present} \to \mathit{StreamAnswer} \\
\mathit{MonitorAnswer} &:= \mathbb{P}(\mathit{Message}) \times (\mathit{done} \ + \ \mathit{next} \ \mathsf{of} \ \mathit{StreamStep}) \\
\mathit{FormulaStep} &:= \mathit{Present} \to \mathit{FormulaAnswer} \\
\mathit{FormulaAnswer} &:= \mathit{done} \ \mathsf{of} \ \mathit{Bool} \ + \ \mathit{next} \ \mathsf{of} \ \mathit{FormulaStep} \\
\mathit{TermStep} &:= \mathit{Present} \to \mathit{TermAnswer} \\
\mathit{TermAnswer} &:= \mathit{done} \ \mathsf{of} \ \mathit{Value} \ + \ \mathit{next} \ \mathsf{of} \ \mathit{TermStep}
\end{aligned}
$$

Different from the previous simplified explanation, the individual engines actually do not only accept as arguments a single message but a "present" p which essentially contains the overall state of all streams (the external as well as the internal ones) at the current step. The reason is that we want e a single copy of the streams' state (rather than each engine creating its own copy) in order to be able by a central optimization to prune this copy as much as possible and thus bound the memory of the computation (see below).

The individual engines now produce the following results:

1. For a monitor engine $M \in \mathit{MonitorStep}$, $M(p)$ denotes the set of violating positions reported in the current step and the next state of the engine.
2. For an internal stream engine $S \in \mathit{StreamStep}$, $S(p)$ denotes the set of messages produced in the current step and the next state of the engine.
3. For a formula engine $F \in \mathit{FormulaStep}$, $F(p)$ either denotes a truth value or the next state of the engine.
4. For a term engine $T \in \mathit{TermStep}$, $T(p)$ either denotes an object or the next state of the engine.

While monitors and streams thus produce answers at every step, formulas and terms produce a result only at the termination of the corresponding engine; until this is the case, these engines produce no answer.

The whole specification is thus translated to a set of interacting engines of above kind; from their composition, the reported violations emerge from the overall engine.

Monitor State and History Pruning The state of the overall monitoring engine consists of the states of the individual sub-engines and of the histories of all streams (see Figure 7); to these histories, in every step new messages may be added from external sources or by the internal stream engines. Without further optimization, for infinite streams eventually the memory of the machine running the monitor would be filled.

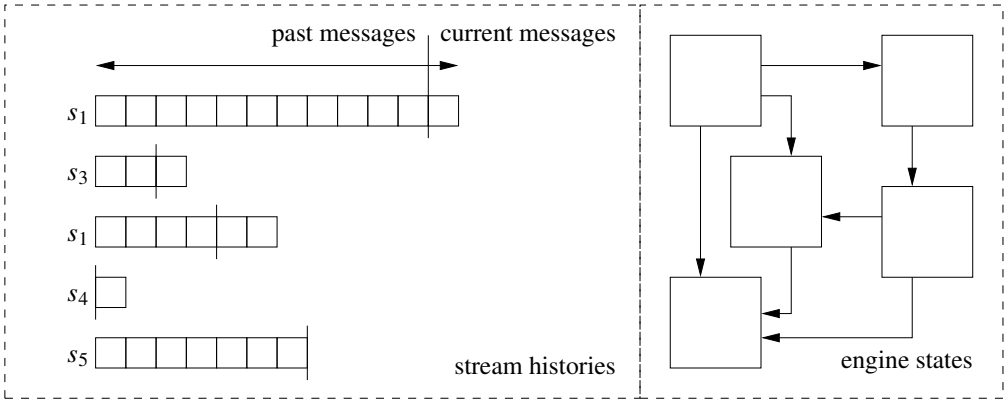

Figure 7: The State of a Monitor

However, the implementation incorporates the analysis and optimization described in [5] which is for certain kinds of specifications able to determine bounds on the number respectively age of messages that need to be preserved in the histories of streams. The analysis passes these bounds to the execution engine which after every execution step (triggered by the arrival of a message on an external stream) prunes the histories of the stream correspondingly; thus perpetual monitors running over infinite streams become possible. Specifications, for which the analysis is not able to determine an appropriate bound are rejected (unless the command line option `-execute` is used, see Section A).

The history pruning analysis succeeds, if in every nested quantification the position variable y introduced by an inner quantifier is related in a particular way to the position variable x introduced by the outermost quantifier (typically a `monitor` or a `stream` quantifier) such as in the specification fragment

```
monitor<S> M =
  monitor<S> x ... :
    ...
    exists<S> y with ... : // with x <(=) _, with x-T <(=)# _
      ...
```

The relationship has to be such that

- both positions x and y refer to the same stream and there exists an upper limit on the position difference $x - y$; this is in particularly true, if the introduction of variable y is accompanied by a clause `with x <= _` or `with x < _` (which indicates that the difference is at most 0); and/or

- positions x and y refer to arbitrary (also different) streams but there exists an upper limit on the time difference $\#x - \#y$ between the time $\#x$ when a message arrived at position x and the time $\#y$ when a message arrived at position y; this is in particular true, if the introduction of variable y is accompanied by a clause `x-T <=# _` or `x-T <# _` (which indicates that the time difference is at most T respectively $T - 1$ time units).

In the first case, to evaluate the inner quantifier no history at all has to be preserved for stream S; in the second case, only those messages have to be preserved that are at most T (respectively $T-1$) time units older than the message that has arrived last on S.

Bounds can also be established in multiple stages via indirect relationships of position variables such as in

```
monitor<S> M =
  monitor<S> x ... :
    exists<S> y with x-100 <# _ : ...
      exists<S> z with y+30 <# : ...
        ...
```

where an upper bound 100 can be established for the time difference $\#x - \#y$ and therefore also an upper bound 70 can be established for the time difference $\#x - \#z$.

In Section 4 we will discuss how specifications with unbounded history requirements that are rejected by the history pruning analyzer can be transformed into logically equivalent specifications that cope with a limited amount of stream history and are thus accepted.

4 Some Sample Specifications

We continue with some more examples on integer streams that elaborate more features of the specification language. We use the following declarations:

```
type int;
stream<int> IP;

logical IsZero(value<int> x);
logical IsOne(value<int> x);
logical IsTwo(value<int> x);
logical NotTooMuch(value<int> x);

value<int> Zero();
value<int> Square(value<int> x);
value<int> Append(value<int> x, value<int> y);
```

Our sample specifications thus check properties of a stream `IP` of `int` values with the help of a couple of externally defined predicates and functions.

Stream Filters In our first example, we transform the external stream `IP` to an internal stream `S` by letting only the ones and the twos pass and by squaring the values that have passed:

```
stream<int> S =
  stream<IP> x
    value<int> m = @x
    satisfying IsOne(m) || IsTwo(m) :
      Square(m);
```

The `stream` quantifier constructs a stream whose values occur in the same order as the positions in stream `IP` over which the bound variable `x` is quantified. For every such position, we define the local variable `m` that denotes the value that `IP` holds at that position; using that variable, we check by the `satisfying` clause whether this value is a one or a two: if yes, then `m` is squared and put onto the result stream.

A corresponding classical set definition would look as follows:

$$S := \{m^2 \mid_{x,m} x \in \mathrm{dom}(IP) \wedge m = IP(x) \wedge (m = 1 \vee m = 2)\}$$

Hhere IP is a function from stream positions to stream values and $\mathrm{dom}(IP)$ denotes the domain of IP, i.e., the set of positions in IP (however, the result S is here defined to be a set of values, not a stream, thus the analogy breaks).

An input stream

$$2,1,0,0,0,2,1,0,2,0,2,0,2,1,2,1,2,0,2,0,2,1,2$$

is thus transformed to the result stream

$$4_{000}, 1_{010}, 4_{050}, \underline{1_{060}}, 4_{080}, 4_{100}, 4_{120}, 1_{130}, 4_{140}, \underline{1_{150}}, 4_{160}, 4_{180}, 4_{200}, \underline{1_{210}}, 4_{220}$$

where the subscripts denote the times when the values appear on `S` (they are the same as the times of the corresponding values on `IP`).

Time Constraints The following specification then checks whether the ones in the previously constructed stream `S` are not more than 50 time units apart:

```
monitor<S> M =
  monitor<S> x :
    IsOne(@x) =>
      exists<S> y with x < _ <=# x+50 : IsOne(@y);
```

In this specification, for every position `x` in `S` (clause `monitor<S> x`) that denotes a one (clause `IsOne(@x)`), it is checked whether there exists a another position `y` in `S` (clause `exists<S>`) where also a 1 occurs (clause `IsOne(@y)`). This position must occur after `x` (clause `x < _`, the underscore `_` denotes the currently quantified variable) but its message must not occur more than 50 time units later than the message at `x` (clause `_ <# x+50`).

In classical predicate logic, we could write a similar definition as follows:

$$M := \{x \in \mathrm{dom}(S) : \neg(S(x) = 1 \Rightarrow \exists y \in \mathrm{dom}(S) : x < y \wedge S_t(y) < S_t(x) + 50 \wedge S(y) = 1)\}$$

Here M denotes the set of all positions that do *not* satisfy the desired property, S is a function from stream positions to stream values, and S_t is a function from stream positions to stream times.

For above example stream, the property is violated for the underlines positions $x = 3, 9, 13$. This is also the output produced by the system:

```
0: 2 # 0
...
12: 2 # 120
VIOLATION<M>: position<S> x=3
13: 1 # 130
...
21: 1 # 210
VIOLATION<M>: position<S> x=9
22: 2 # 220
Message trace is completed.
VIOLATION<M>: position<S> x=13
```

The first two violations are reported when the allowed time bound has been passed, the last one is reported when the stream terminates.

Message Combinations Let us assume we would like to transform a stream `IP` of decimal digits like

$$0,1,2,3,0,4,5,6,7,0,8,0,0$$

to the stream `S2` of numbers

$$123,4567,8,0$$

that i.e. we would like to compute the numbers that result from appending the digits between successive occurrences of zeroes. This can be achieved by the following specification:

```
stream<int> S2 =
  stream<IP> x
    satisfying IsZero(@x) :
    value[seq,Zero(),Append]<IP>
      y with x < _
      value<int> m = @y
      while !IsZero(m) :
        m ;
```

Here the outer `stream` construct binds `x` to all positions of `x` with zero values and then computes the resulting stream value by an application of `value[seq,Zero(),Append]` as follows:

- The construct generates all positions `y` in `IP` that occur after `x` but only as long as the corresponding stream value (which is assigned to a local variable `m`) does not hold a zero. This yields a sequence of stream values

  $$m_1, m_2, \ldots m_n$$

  where m_1 is the value at the position after `x` and m_n is the last value that is not zero.

- From this sequence, the construct computes the value

  `Append(Append(Append(Zero(),` m_1`),` m_2`)` $\ldots$`,` m_n`)`

i.e., starting with the base value `Zero()`, the binary function `Append` is iteratively applied to the previous intermediate value and to the next message value m_i in order to produce the next intermediate value until all messages have been consumed. Provided that `Zero()` denotes zero and `Append(`m`,`n`)` denotes $10m+n$, we may thus compute the number whose representation consists of the digits $m_1,\ldots,m_n$.

Using the auxiliary monitor

```
monitor<S2> PrintS2 =
  monitor<S2> x : PrintValue(@x);
```

the system thus produces for the sample content of `IP` shown above the following output that illustrates the expected contents of `S2`:

```
0: 0 # 0
1: 1 # 10
2: 2 # 20
3: 3 # 30
4: 0 # 40
Value: 123
5: 4 # 50
6: 5 # 60
7: 6 # 70
8: 7 # 80
9: 0 # 90
Value: 4567
10: 8 # 100
11: 0 # 110
Value: 8
12: 0 # 120
Value: 0
```

Quantified Phrases Quantified phrases can be arbitrarily nested. For instance, the monitor

```
monitor<IP> M1 =
  monitor<IP> x :
    IsZero(@x) =>
      exists<IP> y with x < _ <= x+100 :
        IsOne(@y) &&
        forall<IP> z with x < _ < z :
          IsTwo(@x)
```

verifies that every position `x` with value zero is followed within 100 time units by a position `y` that has value one such that between `x` and `y` there are only positions `z` with value two.

There is also a rich variety of non-logical quantifiers. For instance, the monitor

```
monitor<IP> M2 =
  monitor<IP> x :
    IsZero(@x) =>
      value<number> n = num<IP> y with _ < x until IsZero(@y) : true
      NotTooMuch(n)
```

uses the num quantifier to determine the number n of messages that follow a zero message until the next zero message (inclusive) and checks whether NotTooMuch(n) holds (here number denotes the predefined type of the result of num).

The stream definition

```
stream<int> S3 = stream[seq,Zero(),Append]<IP> x : @x;
```

takes an input stream IP of digits

$$1,2,3,4,5,6,\ldots$$

and constructs (for the same definitions of Zero and Append as indicated above) the stream S3 with values

$$0,1,12,123,1234,12345,123456$$

The quantified phrase stream[seq,Zero(),Append] thus behaves like the previously introduced phrase value[seq,Zero(),Append] except that not only the final combination result is returned as a single value but the sequence of all intermediate results is returned as a (potentially ifinite) stream of values.

Unbounded Stream History The monitoring of "backward-oriented" specifications may require an unbounded amount of stream history such that the history analysis rejects the specification. However, frequently such specifications can be easily transformed into logically equivalent ones that are "forward-oriented" such that the analysis succeeds and the specification is accepted for monitoring.

For instance, take a specification of form

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    position<S> y = max<S> y with _ < x : Q(@y) :
    R(@x,@y)
```

which checks all subsequent occurrences of messages with properties Q and P whether they satisfy some common property R. This specification is rejected because the max quantifier looks "backward" arbitrarily far in the stream. However, rewriting the specification as

```
monitor<S> M =
  monitor<S> y : Q(@y) =>
    position<S> x = min<S> x with y < _ : Q(@y) :
    R(@x,@y)
```

we get a characterization of the same property where the min quantifier looks "forward"; this specification copes without stream history and is thus accepted by the analyzer.

However, not all specifications can be transformed in this simple way. For instance, take a specification of form

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    exists<S> y with _ < x : Q(@y) && R(@x,@y)
```

where monitor M checks whether every occurrence of a message with property P is preceded by a message with property Q such that both messages are related by property R. Monitoring this specification would require the preservation of an unlimited amount of stream history; it is thus rejected by the analyzer.

If we could provide a time bound T such that the observation of Q would have to precede that of P by at most T time units, then we could write a specification

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    exists<S> y with x-T <# _ < x : Q(@y) && R(@x,@y)
```

which would be accepted by the history pruning analyzer because only messages have to be preserved that are not more than T time units older than the youngest message.

If, however, no such time bound can be given, the same specification can be still expressed in the more clumsy form

```
stream<T> S0 = stream(F,B,seq)<S> x satisfying P(@x) || Q(@x): @x
monitor<S0> M = monitor<S0> x: R0(@x)
```

where all observations of messages with property P and Q are "registered" in a stream S_0 whose elements of some type T capture all observations of Q and the last observation of P; the external function F creates the next element on that stream from the previous element on the stream and the next message observed. The monitor M checks with some external predicate R_0 every element of that stream whether the last registered observation was a message with property P that was not been preceeded by an observation of a message with property Q such that the two messages are related by property R.

While this is a more "indirect" form of specification that also relies on external functions and predicates, it can nevertheless express the property of interest. In fact, the same technique can be applied to transform any kind of backward looking specification into a forward looking one: we traverse the stream that is monitored and create from all the observations with the help of externally defined functions another stream whose elements capture the information that interests us; this stream is then further processed for violations of certain criteria by externally defined predicates.

5 Future Work

With the current version of the language and its prototypical implementation, we are able to generate from high-level specifications perpetually running monitors that check live network traffic for the violations of complex properties; corresponding applications are going to be reported elsewhere.

The history analysis and pruning technique applied in the prototype makes sure that certain properties can be effectively montitored for an indefinite amount of time, because only a limited amount of stream history is preserved during the execution of the monitor. However, while this is a necessary condition to let the execution of a monitor cope with a bounded amount of memory, it is not a sufficient one. Take e.g. the specification

```
monitor<S> M = monitor<S> x: exists<S> y with x < _ : P(@x,@y)
```

where every message arriving on stream S at some position x gives rise to a new monitor instance that waits for the arrival of another message at a position $y > x$ such that P holds for the pair of messages. If the existence of such a y is not guaranteed, the number of instances that have to be preserved and executed can grow without bound.

Our future work will thus focus on the analysis of specifications with respect to the space complexity of their monitoring such that for those specifications where no upper space bound can be statically determined, the user is warned that monitoring the specification might not be possible with a limited memory budget.

Furthermore, we have not yet addressed the analysis of upper time bounds for the analysis of a specification per incoming message such that we can guarantee certain real-time constraints of the monitor; since the amount of time required for processing a message depends on the number of monitor instances that have to be evaluated, the space complexity analysis sketched above is also a prerequisite for the time complexity analysis.

Moreover, the current evaluation strategy is in those situations very inefficient where for different values of the outermost monitoring variable, the multiple evaluation of inner quantified phrases unnecessarily duplicates work. We will consider combinations of static analysis and runtime optimizations to avoid such repeated evaluations. Ultimately, we plan to investigate static optimizations to translate specifications whose evaluation is inefficient into logically equivalent ones that can be monitored in a more efficient way.

References

[1] RISC Institute, RISC Software, and SecureGUARD. *LogicGuard II. The Optimized Checking of Time-Quantified Logic Formulas with Applications in Computer Security*. 2014. URL: http://www.risc.jku.at/projects/LogicGuard2/.

[2] Temur Kutsia and Wolfgang Schreiner. *LogicGuard Abstract Language*. Tech. rep. 12-08. Johannes Kepler University, Linz, Austria: Research Institute for Symbolic Computation (RISC), 2012. URL: http://www.risc.jku.at/publications/download/risc_4552/2012-LG-abstract-language.pdf.

[3] Temur Kutsia and Wolfgang Schreiner. *Translation Mechanism for the LogicGuard Abstract Language*. Tech. rep. 12-11. Johannes Kepler University, Linz, Austria: Research Institute for Symbolic Computation (RISC), 2012. URL: http://www.risc.jku.at/publications/download/risc_4601/2012-LG-translation-TR.pdf.

[4] Terence Parr. *ANTLR*. 2014. URL: http://www.antlr.org.

[5] Wolfgang Schreiner and Temur Kutsia. *A Resource Analysis for LogicGuard Monitors*. Tech. rep. Johannes Kepler University, Linz, Austria: Research Institute for Symbolic Computation (RISC), 2013. URL: http://www.risc.jku.at/publications/download/risc_4958/typesystem.pdf.

A Running the System

If the program is started as

```
LogicGuard -help
```

it prints the following message which describes its appropriate usage:

```
LogicGuard 1.01 (November 16, 2015)
Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD)
All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard2/
------------------------------------------------------------------------
usage: LogicGuard [<options>] <specification> <dlls>
<specification>: path to monitor specification file
<dlls>: paths to dll files with functions for the monitor
<options>:
  -include <dirs>: <;>-separated list of specification file directories
  -input <file>: path to PCAP or text file for monitoring recorded traffic
  -interface <mac>: MAC address in hex-form HHHHHHHH for live monitoring
  -output <file>: path to log file to which all console output is copied
  -engine <engine>: use monitor execution <engine> which can be one of:
     ip: engine with IP messages captured live or read from PCAP file (default)
     icmp: engine with ICMP messages captured live or read from PCAP file
     dns: engine with DNS messages captured live or read from PCAP file
     dhcp: engine with DHCP messages captured live or read from PCAP file
     int: engine with integer messages read from plain text file
     random: engine generating non-negative random integer messages
     primitive: engine with primitive messages read from plain text file
  -reportnumber <number>: report statistics every <number> messages
  -reporttime <time>: report statistics every <time> ms
  -help | -h | -?: display this help and exit
  -extended: display this help with extended options and exit
Press <ENTER> to continue.
```

The monitor reads its specification from the file ⟨*specification*⟩ (respectively from the files subsequently included from this file); all specification files are looked up in the current working directory, respectively, if the option `-include` is given, from the directories given in the semicolon-separated list ⟨*dirs*⟩. The definitions of the external operations (functions and predicates) are included from the the dynamic link libraries in the list ⟨*dlls*⟩ which must always include the library `External.dll` which is shipped with the distribution. If the option `"-output"` is given, the output printed on the console is also copied to the log file denoted as ⟨*file*⟩.

The (current) runtime engine monitors a single external stream which is named `IP`. If the option `-input` is given, the traffic of this stream is read from the denoted ⟨*file*⟩; if the option ⟨*interface*⟩ is given, the traffic is read live from the interface whose MAC address is denoted by the hexadecimal number ⟨*mac*⟩. The nature of the traffic is determined by the kind of engine that is employed.

- If the option `-engine ip` is given (or if the `-engine` option is omitted), the traffic consists of a sequence of TCP/IP packets; in this case the content of ⟨*file*⟩ must be in the PCAP format. The packets are delivered as values of type `LogicGuard.Engine.Message` whose

`Payload` field has type `LogicGuard.Engine.TCPIP`. These types are documented by the supplementary file `Message.cs` that is shipped with the distribution.

- If the command line option `-engine -icmp` is given, the traffic consists of a sequence of ICMP packets; also in this case the content of ⟨*file*⟩ must be in the PCAP file format. The packets are delivered as stream values of the type `LogicGuard.Engine.Message` whose `Payload` field is of type `LogicGuard.Engine.ICMP`. These types are documented by the supplementary file `Message.cs` that is shipped with the distribution.

- If the command line option `-engine int` is given, a plain text file is read which consists of a sequence of white-space separated

  ⟨*value*⟩ ⟨*time*⟩

  pairs where ⟨*value*⟩ denotes by an decimal integer literal a stream value (an integer of the .NET type `System.Int64`) and ⟨*value*⟩ denotes by a non-negative decimal integer literal the corresponding time. This engine is used for debugging and testing the system.

- If the command line option `-engine random` is given, a sequence of integer messages is produced by a random number generator. By default, this sequence is infinite (thus the engine runs forever) and contains values in the range $0 \ldots 100$. Also this engine is used for debugging and testing the system; its behavior can be further configured by some extended options listed below.

- If the command line option `-engine dns` is given, the traffic consists of a sequence of DNS packets; also in this case the content of ⟨*file*⟩ must be in the PCAP file format. The packets are delivered as stream of values of the type `LogicGuard.Engine.Message` whose `Payload` field is of type `LogicGuard.Engine.DnsPacket`. These types are provided by the file `Message.cs` that is shipped with distribution.

- If the command line option `-engine dhcp` is given, the traffic consists of a sequence of DHCP packets; also in this case the content of ⟨*file*⟩ must be in the PCAP file format. The packets are delivered as stream of values of the type `LogicGuard.Engine.Message` whose `Payload` field is of type `LogicGuard.Engine.DhcpPacket`. These types are provided by the file `Message.cs` that is shipped with distribution.

- If the command line option `-engine primitive` is given, a plain text file is read which consists of a sequence of white-space separated ⟨*value*⟩ ⟨*time*⟩ pairs: ⟨*value*⟩ denotes a value of the type indicated by the option `-ptype` (see the extended options below) and ⟨*time*⟩ denotes by a non-negative decimal integer literal the corresponding time. This engine is used for debugging and testing the system. The messages are delivered as stream values of the type `LogicGuard.Engine.Message` whose `Payload` field has type `LogicGuard.Engine.PrimitiveValue`. These types are documented in the supplementary file `Message.cs` that is shipped with distribution.

If the command line option `-reportnumber` is provided, the monitor reports every ⟨*number*⟩ messages some runtime statistics; if the option `-reporttime` is provided, the monitor reports

approximately every ⟨*time*⟩ time units some runtime statistics (actually, the statistics is only reported after the first message has been processed that has arrived at least ⟨*time*⟩ units after the last report).

If the program is started as

```
LogicGuard -extended
```

also the following options are printed:

```
extended <options>:
  -stop <step>: stop program after <step> which can be one of:
     parse typecheck inline analyze execute
  -print: print abstract syntax tree of specification
  -pinline: print abstract syntax tree of specification after inlining
  -panalysis: print result of history analysis
  -noprune: prevent history pruning
  -nobinder: disable binder optimization during history analysis
  -execute: force execution even if history analysis shows unbounded history
  -verbose: print messages during monitoring
  -randnumber <number>: <random> generates <number> numbers (default infinity)
  -randtime <time>: <random> runs for <time> ms (default infinity)
  -randbound <bound>: <random> produces numbers up to <bound> (default 100)
  -randdelay <delay>: <random> produces numbers every <delay> ms (default 0)
  -buffer <number>: interface buffer may hold <number> messages (default 25000)
  -timestream: enable time stream in ip live engine
  -tsinterval <time> generate null message if none in <time> ms (default 100)
  -ptype <type>: primitive <type> if primitive engine is used (default Int32)
Press <ENTER> to continue.
```

The extended options are mainly used for development and debugging purposes and should not be needed in normal operation. They allow to terminate the monitor after a certain processing step (option `-stop`), to print certain system information (`-print`, `-pinline`, `-panalysis`), to control certain optimizations (`-noprune`, `-nobinder`, `-execute`), and to display all processed traffic (`-verbose`).

Furthermore, the behavior of the random engine can be configured by the following options:

- If the option `-randnumber` is provided, the engine stops after ⟨*number*⟩ messages have been generated.
- If the option `-randtime` is provided, the engine stops execution after ⟨*time*⟩ milliseconds.
- If the option `-randbound` is provided, the engine produces values in the range 0 to *nm*⟨*bound*⟩.
- By default, messages are produced as fast as possible. However, if the option `-randdelay` is provided, then the mean time between the arrival of two messages becomes ⟨*delay*⟩ ms (in more detail, the arrival of messages follows an exponential distribution with parameter $\lambda = 1/\langle delay \rangle$).

The live network interface buffers incoming messages before they are processed. The command line option `-buffer` determines the maximum ⟨*number*⟩ of messages that the buffer may hold; if the buffer becomes full, additional messages are dropped (with a corresponding warning being printed).

The behavior of the primitive engine can be configured by the option `-ptype` whose ⟨*type*⟩ argument defines the primitive data type of the value read from the file. The default value is `Int32` (`"System.Int32"`). Acceptable types are `"Boolean"`, `"Byte"`, `"SByte"`, `"Char"`, `"Decimal"`, `"Double"`, `"Single"`, `"Int32"`, `"UInt32"`, `"Int64"`, `"UInt64"`, `"Int16"`, as well as `"UInt16"` and `"String"`.

The command line option `"-timestream"` produces auxiliary messages in network live engines: if the engine has not captured any packet during the time interval specified by the option `-tsinterval` (in milliseconds, default is 100 ms), a message is generated whose payload is `null` and whose time stamp is the current system time. This feature works for the live engines `ip`, `icmp`, `dns` and `dhcp`. If this option is enabled, the external functions should be aware of the possibility of `null` values in messages.

B Lexical and Syntactic Analysis

In this section, we describe the *lexical analysis* of a specification file, i.e., how the content of the file is transformed into a sequence of tokens. This transformation proceeds in the three steps of

1. processing include files,
2. processing comments, and
3. tokenization,

which are described in the following subsections.

In the later sections, we will then describe the *syntactic analysis* of a specification, i.e., which sequences of tokens generated by the lexical analysis represent correct specifications. This analysis is defined by a context-free grammar in the Extended Backus Naur Form (EBNF). Such a grammar consists of production rules of form

⟨*symbol*⟩ ::= *alternative* | . . . | *alternative*

where in every *alternative* an occurrence of

[*phrase*]

denotes zero or one occurrence of *phrase* (an *option*) while an occurrence

{ *phrase* }

denotes arbitrary many (zero, one, or more) occurrences of *phrase* (a *repetition*).

B.1 Include Files

On the first level, the content of a specification file is considered as a sequence of lines. If a line contains the string `#include`, it must have one of the forms

```
// #include "⟨path⟩"
#include "⟨path⟩"
```

with only white space before and after this form.

A line of the first form is removed from the content of the file.

In a line of the second form, ⟨*path*⟩ must denote the file system path of another specification file. The content of this file is recursively processed in the same way and then replaces the `#include` line. It is an error, if a file includes itself directly or indirectly recursively. If the same file is included multiple times (from the same file or from different files), it is only included at the first occurrence of the corresponding `#include`; all subsequent occurrences are removed.

Every ⟨*path*⟩ is first considered relative to the current working directory of the system; if the specified file is not found there and if the command line option `-include` ⟨*paths*⟩ is given (see Section A), then each path in ⟨*paths*⟩ is prepended to ⟨*path*⟩; the first file that is found in this way is read. It is an error, if no file can be found by this procedure.

B.2 Comments

On the second level (after include files have been processed), all comments are removed from the content of a specification file.

A comment may be a single-line comment of form

```
// ⟨anytext⟩
```

where ⟨*anytext*⟩ is an arbitrary sequence of characters terminated by the end of the line or the end of the file.

A comment may also be a multi-line comment of form

```
/* ⟨anytext⟩
*/
```

where ⟨*anytext*⟩ is an arbitrary sequence of characters terminated by the fist occurrence of the token "`*/`" (thus multi-line comments cannot be nested). It is an error, if there is no such occurrence, i.e., if the file ends prematurely.

B.3 Tokens

On the third and final level (after comments have been removed), the content of the specification file is transformed into a sequence of tokens. A token is either a terminal symbol of non-whitespace characters (such symbols are subsequently depicted in `teletype`) or one of the following two kinds of symbols whose constructions are depicted by regular expressions:

⟨*ID*⟩ ::= `[a-zA-Z][a-zA-Z_0-9]*`

⟨*TIME*⟩ ::= `[0-9]+`

Thus an *identifier* ⟨*ID*⟩ is a sequence of (upper- and lower-case) letters, decimal digits, and the underscore character "_" which starts with a letter. A *time literal* ⟨*TIME*⟩ is a non-empty sequence of decimal digits.

C Type System

From Section D on, we describe the context-free grammar of the specification language. However, only those specifications are legal that also conform to the static type system of the language.

For the subsequent explanation, we define the auxiliary non-terminal symbols

⟨*tid*⟩ ::= ⟨*ID*⟩

⟨*sid*⟩ ::= ⟨*ID*⟩

for an identifier ⟨*tid*⟩ that denotes a type and an identifier ⟨*sid*⟩ that denotes a stream. Then, in a nutshell, the type system assigns to every term (see Section G) one of the following types:

- `value`<*tid*>: the term represents a value whose type has name *tid*.
- `stream`<*tid*>: the term represents a stream whose values are of the type with name *tid*.
- `position`<*sid*>: the term represents a position in the stream with name *sid*.

Furthermore, every monitor (see Section H) has the following type:

- `monitor`<sid_1,...,sid_n>: the monitor surveys the streams with names sid_1,...,sid_n and reports violations as vectors of positions of these streams.

The type checker verifies that the usages of terms and monitors are consistent with the declarations given by the user and rejects a specification, if this is not the case.

D Specifications

Grammar

⟨*specification*⟩ ::= { [⟨*declaration*⟩] `;` }

Description A specification is a sequence of declarations where every declaration is terminated by a semicolon "`;`". Multiple occurrences of "`;`" without an intervening declaration are also legal.

Every declaration introduces an identifier and assigns it a meaning. This identifier is known in all subsequent declarations.

E Declarations

Grammar

⟨*declaration*⟩ ::=
    `type` ⟨*ID*⟩
  | `logical` ⟨*ID*⟩ = ⟨*formula*⟩
  | `logical` ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
  | `logical` ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*formula*⟩
  | `value` < ⟨*tid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
  | `value` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
  | `value` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩
  | `position` < ⟨*sid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
  | `position` < ⟨*sid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩
  | `stream` < ⟨*tid*⟩ > ⟨*ID*⟩
  | `stream` < ⟨*tid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
  | `stream` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
  | `stream` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩
  | `monitor` < [⟨*sid*⟩ { `,` ⟨*sid*⟩}] > ⟨*ID*⟩ = ⟨*monitor*⟩

Description Every declaration introduces an identifier ⟨*ID*⟩ and assigns it a meaning. There are four separate identifier namespaces for the following four kinds of entities:

- Types,
- Logical entities,
- Objects (values, positions, streams),
- Monitors.

The grammar can distinguish whether a particular occurrence of an identifier must denote a type, a logical entity, an object, or a monitor. Therefore we may declare a type, a logical entity, an object, and a monitor all with the same identifier. However, it is an error to have two declarations for the same kind of entity with the same identifier.

In the following subsections, we describe the various kinds of declarations in more detail.

Pragmatics All declarations that define a value start their evaluations simultaneously with the first message arriving on an external stream. In other words, it is not the case that the final value of a definition must be available before the evaluations of the subsequent definitions can start.

E.1 Type Declarations

Grammar

`type` ⟨*ID*⟩

Description This declaration introduces a new type with name ⟨*ID*⟩ that is different from all other types.

There are two predefined types (that cannot be declared by the user):

- type `time` which denotes the type of message time terms (see Section G.1).
- type `number` which denotes the type of number terms (see Section G.4).

Pragmatics Within this specification, it is unknown how a user-declared type is externally represented. However, the runtime system assumes that a value of this type can be converted from/to the .NET type `System.Object` (which is true for all .NET types).

Example The declaration

```
type int;
```

introduces a type `int` (which may or may not be mapped to the .NET type `System.Int64`).

E.2 Logical Declarations

Grammar

`logical` ⟨*ID*⟩ = ⟨*formula*⟩
`logical` ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
`logical` ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*formula*⟩

Description These declarations introduce a new logical entity with name ⟨*ID*⟩ from which a logical formula can be constructed:

1. In the first declaration, a logical variable is introduced and defined to be equivalent to the given ⟨*formula*⟩; an occurrence ⟨*ID*⟩ of this variable is a formula that denotes the truth value of the defining ⟨*formula*⟩.
2. In the second declaration, an externally defined parameterized predicate is introduced; an application ⟨*ID*⟩(...) with the number and kinds of arguments that correspond to the given parameters (see Section K) represents an "atomic formula".
3. In the third declaration, also a parameterized predicate is introduced but internally defined by the given formula; an application ⟨*ID*⟩(⟨*term*⟩, ...) with the necessary number and kinds of arguments is a formula that denotes the truth value of the defining ⟨*formula*⟩ after substitution of the formal parameters by the actual arguments.

Pragmatics The value of a logical variable is only computed once and shared by all occurrences of this variable. The value of an externally defined predicate is computed separately for each application of the predicate by calling an external function with the same name whose result type is the .NET type `System.Boolean`. Every application of an internally defined predicate is inlined by substituting the appropriately instantiated formula for the application (the instantiation makes sure that no argument is evaluated more than once).

Example The declaration

```
logical P(value<int> x) = forall<IP> y : R(x, @y);
```

introduces an internally defined predicate *P* that takes a single value of type `int` as argument.

E.3 Value Declarations

Grammar

`value` < ⟨*tid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
`value` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
`value` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩

Description These declarations introduce a new value variable or value function with name ⟨*ID*⟩ and (result) type ⟨*tid*⟩ from which a term can be constructed:

1. In the first declaration, a value variable is introduced and defined to be equal to the given ⟨*term*⟩ (which must denote a value of type ⟨*tid*⟩); an occurrence ⟨*ID*⟩ of this variable is a term that denotes the value of the defining ⟨*term*⟩.
2. In the second declaration, an externally defined value function is introduced; an application ⟨*ID*⟩(...) with the number and kinds of arguments that correspond to the given parameters (see Section K) represents a "function application".
3. In the third declaration, also a value function is introduced but internally defined by the given ⟨*term*⟩ (which must denote a value of type ⟨*tid*⟩); an application ⟨*ID*⟩(...) with the necessary number and kinds of arguments is a term that denotes the value of the defining ⟨*term*⟩ after substitution of the formal parameters by the actual arguments.

Pragmatics The content of a value variable is only computed once and shared by all occurrences of this variable. The value of an externally defined value function is computed separately for each application of the function by calling an external function with the same name whose result type can be converted from/to the .NET type `System.Object`. Every application of an internally defined function is inlined by substituting the appropriately instantiated term for the application (the instantiation makes sure that no argument is evaluated more than once).

Example The declaration

```
value<int> F(value<int> x) = if P(x) then x else G(x);
```

introduces an internally defined function *F* that takes a single value of type `int` as argument.

E.4 Position Declarations

Grammar

`position` < ⟨*sid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
`position` < ⟨*sid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩

Description These declarations introduce a new position variable or position function with name ⟨*ID*⟩ whose (result) value denotes a position in stream ⟨*sid*⟩ from which a term can be constructed:

1. In the first declaration, a position variable is introduced and defined to be equal to the given ⟨*term*⟩ (which must denote a position in stream ⟨*sid*⟩); any occurrence ⟨*ID*⟩ of this variable then is a term that denotes the value of the defining ⟨*term*⟩.

2. In the second declaration, a position function is introduced and internally defined by the given ⟨*term*⟩ (which must denote a position in stream ⟨*sid*⟩); an application ⟨*ID*⟩(...) with the number and kinds of arguments that correspond to the given parameters (see Section K) is a term that denotes the value of the defining ⟨*term*⟩ after substitution of the formal parameters by the actual arguments.

Pragmatics The value of a position variable is only computed once and shared by all occurrences of this variable. Every application of an internally defined position function is inlined by substituting the appropriately instantiated term for the application (the instantiation makes sure that no argument is evaluated more than once).

Example The declaration

```
position<IP> p = max<IP> p with q < _ <# q+1000: P(@p)
```

introduces a position variable `p` which denotes the largest position in stream `IP` such that `P(@p)` holds where `p` occurs after position `q` but before 1000 time units after `q`.

E.5 Stream Declarations

Grammar

`stream` < ⟨*tid*⟩ > ⟨*ID*⟩
`stream` < ⟨*tid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
`stream` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)`
`stream` < ⟨*tid*⟩ > ⟨*ID*⟩ `(` [⟨*parameter*⟩ { `,` ⟨*parameter*⟩ }] `)` = ⟨*term*⟩

Description These declarations introduce a new stream variable or stream function with name ⟨*ID*⟩ and message type ⟨*tid*⟩ from which a term may be constructed:

1. In the first declaration, an externally defined stream variable is introduced; an occurrence ⟨*ID*⟩ of this variable is a term that denotes this stream.

2. In the second declaration, an internal stream variable is introduced and defined to be equal to the given ⟨*term*⟩ (which must denote a stream with message type ⟨*tid*⟩); an occurrence ⟨*ID*⟩ of this variable is a term that denotes the value of the defining ⟨*term*⟩.

3. In the third declaration, an externally defined stream function is introduced; an application ⟨*ID*⟩(...) with the number and kinds of arguments that correspond to the given parameters (see Section K) is a term.

4. In the fourth declaration, also a stream function is introduced but internally defined by the given term (which must denote a stream with message type ⟨*tid*⟩); an application ⟨*ID*⟩(...) with the necessary number and kinds of arguments is a term that denotes the value of the defining ⟨*term*⟩ after substitution of the formal parameters by the actual arguments.

Pragmatics An external stream variable is filled by the runtime environment. The content of an internal stream variable is only computed once and shared by all occurrences of this variable. Every application of an internally defined function is inlined by substituting the appropriately instantiated term for the application (the instantiation makes sure that no argument is evaluated more than once).

The value of an externally defined stream function is computed separately for each application of the function in the following way: at every execution step, an external function with the same name is called whose result type must be convertible from/to the .NET type `System.Object[]`. If the function returns a non-`null` result, this result denotes the set of messages that are delivered by the stream in the current step. If the function returns `null`, this indicates that the stream has terminated and the function must not be called any more for further messages.

Example The declaration

```
stream<int> S = stream<int> x in IP satisfying P(@x) : @x
```

introduces an internally defined stream variable `S` that contains those messages from stream `IP` that satisfy the predicate `P`.

E.6 Monitor Declarations

Grammar

`monitor` < [⟨*sid*⟩ { `,` ⟨*sid*⟩}] > ⟨*ID*⟩ = ⟨*monitor*⟩

Description This declaration introduces a monitor with name ⟨*ID*⟩ that monitors the streams ⟨*sid*⟩, ...; the monitor is defined by the monitor body ⟨*monitor*⟩ which must monitor the listed streams.

At each step of the execution the monitor produces a set of

- *violating positions*, i.e., positions for which the monitoring formula has yielded the truth value “false”, and
- *warning positions*, i.e., positions for which the monitoring formula has yielded the truth value “unknown”.

Example The monitor

```
monitor<S1,S2> M = monitor<S1> x: monitor<S2> y: P(@x, @y);
```

monitors two stream `S1` and `S2`; it reports all position pairs `x` and `y` as violations for which the predicate `P(@x,@Y)` returns “false”.

F Formulas

Grammar

⟨*formula*⟩ ::=
    `(` ⟨*formula*⟩ `)`
  | `true`
  | `false`
  | `logical ?`
  | `defined` ⟨*term*⟩
  | `defined` ⟨*formula*⟩
  | ⟨*ID*⟩
  | ⟨*ID*⟩ `(` `[` ⟨*term*⟩ `{` `,` ⟨*term*⟩ `}` `]` `)`
  | `!` ⟨*formula*⟩
  | ⟨*formula*⟩ `&&` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩
  | ⟨*formula*⟩ `||` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩
  | ⟨*formula*⟩ `=>` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩
  | ⟨*formula*⟩ `<=>` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩
  | `if` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩ `then` ⟨*formula*⟩ `else` ⟨*formula*⟩
  | `forall` ⟨*variable*⟩ ⟨*formula*⟩
  | `exists` ⟨*variable*⟩ ⟨*formula*⟩
  | ⟨*binder*⟩ `:` ⟨*formula*⟩

Description A formula denotes a logical entity with truth values “true”, “false”, and “unknown”. The value “unknown” indicates that some term on which the value of the formula depends (see Section G) has no meaningful value. The formulas are listed above in the decreasing order of the binding power of their operators; in particular, `!` binds stronger than `&&` which binds stronger than `||`.

F.1 Atomic Formulas

Grammar

`(` ⟨*formula*⟩ `)` `true`
`false`
`logical ?`
`defined` ⟨*term*⟩
`defined` ⟨*formula*⟩

⟨*ID*⟩
⟨*ID*⟩ `(` [⟨*term*⟩ { `,` ⟨*term*⟩ }] `)`

Description The parenthesized formula `(` ⟨*formula*⟩ `)` just stands for ⟨*formula*⟩ itself. The formulas `true`, `false`, and `logical ?` denote the logical values "true", "false", and "unknown", respectively. The formula `defined` ⟨*term*⟩ denotes "true", if ⟨*term*⟩ does not denote "undefined", and "false", otherwise. Likewise, `defined` ⟨*formula*⟩ denotes "true", if ⟨*formula*⟩ does not denote "undefined", and "false", otherwise.

The formula ⟨*ID*⟩ denotes the value of the logical variable ⟨*ID*⟩. The formula ⟨*ID*⟩ `(` [⟨*term*⟩ { `,` ⟨*term*⟩ }] `)` denotes the value of the internally or externally defined predicate ⟨*ID*⟩ when applied to the values of the given terms.

Example The atomic formula

```
R(x, @y)
```

denotes the value of the binary predicate *R* when applied to the two given arguments.

F.2 Propositional Formulas

Grammar

`!` ⟨*formula*⟩
⟨*formula*⟩ `&&` [`[` ⟨*mode*⟩ `]`] ⟨*formula*⟩
⟨*formula*⟩ `||` [`[` ⟨*mode*⟩ `]`] ⟨*formula*⟩
⟨*formula*⟩ `=>` [`[` ⟨*mode*⟩ `]`] ⟨*formula*⟩
⟨*formula*⟩ `<=>` [`[` ⟨*mode*⟩ `]`] ⟨*formula*⟩

Description

1. The unary operator `!` denotes logical negation as described by the following table (where ⊤ denotes "true", ⊥ denotes "false" and ? denotes "unknown"):

⟨*formula*⟩	`!` ⟨*formula*⟩
⊤	⊥
⊥	⊤
?	?

2. The binary operators `&&`, `||`, `=>`, `<=>` denote logical conjunction, disjunction, implication, and equivalence as described by the following table:

⟨*f1*⟩	⟨*f2*⟩	⟨*f1*⟩ && ⟨*f2*⟩	⟨*f1*⟩ || ⟨*f2*⟩	⟨*f1*⟩ => ⟨*f2*⟩	⟨*f1*⟩ <=> ⟨*f2*⟩
⊤	⊤	⊤	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊥	⊥
⊤	?	?	⊤	?	?
⊥	⊤	⊥	⊤	⊤	⊥
⊥	⊥	⊥	⊥	⊤	⊤
⊥	?	⊥	?	⊤	?
?	⊤	?	⊤	⊤	?
?	⊥	⊥	?	?	?
?	?	?	?	?	?

If the optional qualifier `[`⟨*mode*⟩`]` is not given or if it is `[par]`, both parts of the formula are evaluated simultaneously. If the qualifier is given as `[seq]`, then first the first formula is evaluated. If the result already uniquely determines the overall result, the second formula is not evaluated ("short-circuit evaluation").

Pragmatics The default evaluation mode of most propositional formulas is "parallel" to avoid unnecessary delays in formula evaluations and the corresponding need for message buffering. The "sequential" mode is provided mainly to allow "short-circuited evaluation" as is supported in many programming languages where the evaluation of the second formula is only allowed in a context where the first formula does not determine the result (i.e., if the evaluation of the second formula might yield a runtime error, otherwise).

Example The formula

```
R(x) => P(x,y)
```

denotes an implication evaluated in the default "parallel" mode.

F.3 Conditional Formulas

Grammar

`if [ [` ⟨*mode*⟩ `] ]` ⟨*formula0*⟩ `then` ⟨*formula1*⟩ `else` ⟨*formula2*⟩

Description This formula denotes conditional selection: if ⟨*formula0*⟩ yields "true", its value is that of ⟨*formula1*⟩; if ⟨*formula0*⟩ yields "false", its value is that of ⟨*formula2*⟩; if ⟨*formula0*⟩ yields "unknown", its value is "unknown".

If the optional qualifier `[`⟨*mode*⟩`]` is not given, or if it is `[seq]` then ⟨*formula0*⟩ is evaluated first to decide which of ⟨*formula1*⟩ or ⟨*formula2*⟩ is to be evaluated (i.e., only one of the formulas is evaluated). If the qualifier is given as `[par]`, all formulas are evaluated in parallel; only when the evaluation of ⟨*formula0*⟩ has completed, the remainder of the evaluation proceeds with only one of ⟨*formula1*⟩ or ⟨*formula2*⟩.

Pragmatics The default evaluation for the conditional formula is "sequential" to support the usual intuition that only one of the branches is evaluated; however, the "parallel" mode should be considered to avoid unnecessary evaluation delays and corresponding stream buffering requirements.

Example The formula

```
if P(x) then Q(x) else R(x)
```

essentially corresponds to the classical formula

$$(P(x) \Rightarrow Q(x)) \wedge (\neg P(x) \Rightarrow R(x))$$

F.4 Quantified Formulas

Grammar

`forall` ⟨*variable*⟩ ⟨*formula*⟩
`exists` ⟨*variable*⟩ ⟨*formula*⟩

Description In both formulas, the body ⟨*formula*⟩ is evaluated in all contexts where the local position variable introduced by ⟨*variable*⟩ is bound to the positions denoted by ⟨*variable*⟩ (see Section L).

1. In the case of `forall`, if there is some instance of the formula body which yields "false", the result is "false". If there is no such instance but an instance that yields "unknown", the result is "unknown". Otherwise (i.e., if all instances yield "true"), the result is "true".
2. In the case of `exists`, if there is some instance of the formula body which yields "true", the result is "true". If there is no such instance but an instance that yields "unknown", the result is "unknown". Otherwise (i.e., if all instances yield "false"), the result is "false".

Pragmatics Both the elaboration of new formula instances and the evaluation of already created instances proceeds simultaneously (for more details, see Section L).

Example The formula

```
forall<IP> x with y <= _ <=# x + 1000 : P(@x)
```

is true if the formula `P(@x)` is true for all positions `x` in stream `IP` starting with position `y` and ending with the largest position that occurs not more than 1000 time units later than y.

F.5 Formulas with Local Bindings

Grammar

⟨*binder*⟩ `:` ⟨*formula*⟩

Description The body ⟨*formula*⟩ is evaluated in a context where a new variable has been introduced and its value been defined by ⟨*binder*⟩ (see Section J).

Pragmatics The evaluation of the defining phrase in ⟨*binder*⟩ and that of ⟨*formula*⟩ proceed simultaneously; in order words, it is not the case that the variable introduced in ⟨*binder*⟩ must have its ultimate value before the evaluation of ⟨*formula*⟩ can start.

Example The formula

```
value<int> y = F(@x) : P(y)
```

introduces a local value `y` defined as `F(@x)` for the evaluation of `P(y)`.

G Terms

Grammar

⟨*term*⟩ ::=
`(` ⟨*term*⟩ `)`
| `value` ⟨*tid*⟩ `?`
| `position` ⟨*sid*⟩ `?`
| `stream` ⟨*tid*⟩ `?`
| `zero` ⟨*sid*⟩
| `empty` ⟨*tid*⟩
| `old`
| `new`
| ⟨*ID*⟩
| ⟨*ID*⟩ `(` `[` ⟨*term*⟩ `{` `,` ⟨*term*⟩ `}` `]` `)`
| `[` ⟨*ID*⟩ `]` `@` ⟨*term*⟩
| `[` ⟨*ID*⟩ `]` `#` ⟨*term*⟩
| `if` `[` `[` ⟨*mode*⟩ `]` `]` ⟨*formula*⟩ `then` ⟨*term*⟩ `else` ⟨*term*⟩
| `min` ⟨*variable*⟩ ⟨*formula*⟩
| `max` ⟨*variable*⟩ ⟨*formula*⟩
| `num` ⟨*variable*⟩ ⟨*formula*⟩
| `value` `[` ⟨*mode2*⟩ `,` ⟨*term1*⟩ `,` ⟨*ID*⟩ `]` ⟨*variable*⟩ ⟨*term2*⟩
| `stream` `[` ⟨*mode*⟩ `]` ⟨*variable*⟩ ⟨*term*⟩
| `stream` `[` ⟨*mode2*⟩ `,` ⟨*term1*⟩ `,` ⟨*ID*⟩ `]` ⟨*variable*⟩ ⟨*term2*⟩
| `merge` `[` ⟨*mode*⟩ `]` ⟨*variable*⟩ ⟨*term*⟩
| ⟨*binder*⟩ `:` ⟨*term*⟩

Description A term denotes an object whose values depend on the kind of term (terms may denote values, positions, or streams). In any case, the value of a term may be “undefined”. The terms are listed above in the decreasing order of the binding power of their operators.

G.1 Basic Terms

Grammar

`(` ⟨*term*⟩ `)`
`value` ⟨*tid*⟩ `?`
`position` ⟨*sid*⟩ `?`
`stream` ⟨*tid*⟩ `?`
`zero` ⟨*sid*⟩
`empty` ⟨*tid*⟩
`old`
`new`
⟨*ID*⟩
⟨*ID*⟩ `(` [⟨*term*⟩ { `,` ⟨*term*⟩ }] `)`
[⟨*ID*⟩] `@` ⟨*term*⟩
[⟨*ID*⟩] `#` ⟨*term*⟩

Description The parenthesized term `(` ⟨*term*⟩ `)` just stands for ⟨*term*⟩ itself. The constant `value` ⟨*tid*⟩ `?` denotes the "unknown" value of type ⟨*tid*⟩; `position` ⟨*sid*⟩ `?` denotes the "unknown" position of stream ⟨*sid*⟩; `stream` ⟨*tid*⟩ `?` denotes the "unknown" stream with values of type ⟨*tid*⟩. The constant `zero` ⟨*sid*⟩ denotes the initial position in stream ⟨*sid*⟩; `empty` ⟨*tid*⟩ denotes the empty stream with values of type ⟨*tid*⟩.

The constant `old` may only occur in the `until` or `while` formula of a combining `value` or `stream` term with evaluation mode `[strict]` (see Sections G.4 and G.5). It denotes the previous value of the combination (before the combining function integrates the next value).

The constant `new` may only occur in the `until` formula of a combining `value` or `stream` term with evaluation mode `[strict]` (see Sections G.4 and G.5). It denotes the current value of the combination (after the combining function has integrated the next value).

The variable ⟨*ID*⟩ denotes the object (value, position, stream) to which it is assigned in the current context.

The function application ⟨*ID*⟩ `(` [⟨*term*⟩ { `,` ⟨*term*⟩ }] `)` denotes the result of the application of the (value, position, stream) function to which it is assigned in the current context.

In the message value term [⟨*ID*⟩] `@` ⟨*term*⟩, ⟨*ID*⟩ (if given) must denote a stream and ⟨*term*⟩ must denote a position in that stream. The term then denotes the message that the stream holds at that position.

In the message time term [⟨*ID*⟩] `#` ⟨*term*⟩, ⟨*ID*⟩ (if given) must denote a stream and ⟨*term*⟩ must denote a position in that stream. The term then denotes the time of the message that the stream holds at that position (a value of the predefined type `time`).

Pragmatics The constant `zero` ⟨*sid*⟩ may be used to denote in a quantified variable's upper bound (see Section L) by the phrase `zero` ⟨*sid*⟩ + ⟨*TIME*⟩ the time of ⟨*TIME*⟩ units after the time of the initial message on stream ⟨*sid*⟩.

The type `time` is externally mapped to the .NET type `System.Int64`. The unit of time depends on the runtime system; it typically is 100 ns (i.e, 10^7 time units take 1 s).

Example The term

```
@x
```

denotes the message at position `x` in that stream that is referenced by `x`. If that stream is `IP`, then

```
IP@x
```

means the same.

G.2 Conditional Terms

Grammar

`if` [[⟨*mode*⟩]] ⟨*formula*⟩ `then` ⟨*term1*⟩ `else` ⟨*term2*⟩

Description In this term which denotes conditional selection both ⟨*term1*⟩ and ⟨*term2*⟩ must denote the same kind and type of object: if ⟨*formula*⟩ yields "true", its value is that of ⟨*term1*⟩; if ⟨*formula*⟩ yields "false", its value is that of ⟨*term2*⟩; if ⟨*formula*⟩ yields "unknown", its value is "unknown".

If the optional qualifier [⟨*mode*⟩] is not given, or if it is `[seq]` then ⟨*formula*⟩ is evaluated first to decide which of ⟨*term1*⟩ or ⟨*term2*⟩ is to be evaluated (i.e., only one of the terms is evaluated). If the qualifier is given as `[par]`, the formula and the terms are all evaluated in parallel; only when the evaluation of the formula has completed, the remainder of the evaluation proceeds with only one of the terms.

Pragmatics The default evaluation for the conditional term is "sequential" to support the usual intuition that only one of the branches is evaluated; however, the "parallel" mode should be considered to avoid unnecessary evaluation delays and corresponding stream buffering requirements.

Example The term

```
if P(x) then F1(x) else F2(x)
```

returns one of the values `F1(x)` or `F2(x)`.

G.3 Quantified Position Terms

Grammar

`min` ⟨*variable*⟩ ⟨*formula*⟩
`max` ⟨*variable*⟩ ⟨*formula*⟩

Description In both terms, the body ⟨*formula*⟩ is evaluated in all contexts where the local position variable introduced by ⟨*variable*⟩ is bound to all positions denoted by ⟨*variable*⟩ (see Section L).

1. In the case of `min`, if there is some position for which the formula body yields "true" and for all smaller positions yields "false", the term denotes this position. Otherwise the term denotes "unknown".
2. In the case of `max`,i f there is some position for which the formula body yields "true" and for all bigger positions yields "false", the term denotes this position. Otherwise the term denotes "unknown".

Pragmatics Both the elaboration of new formula instances and the evaluation of already created instances proceeds simultaneously (for more details, see Section L).

Example The term

```
min<IP> x with y <= _ <=# x + 1000 : P(@x)
```

denotes the smallest position `x` in stream `IP` for which P(@x) holds where `x` is in the range starting with position `y` and ending with the largest position that occurs not more than 1000 time units later than y.

G.4 Quantified Value Terms

Grammar

`num` ⟨*variable*⟩ ⟨*formula*⟩
`value [` ⟨*mode2*⟩ `,` ⟨*term1*⟩ `,` ⟨*ID*⟩ `]` ⟨*variable*⟩ ⟨*term2*⟩

Description In both terms, the body phrase (⟨*formula*⟩ respectively ⟨*term2*⟩) is evaluated in all contexts where the local position variable introduced by ⟨*variable*⟩ is bound to all positions allowed by the constraints in ⟨*variable*⟩ (see Section L).

In the case of `num`, the term denotes the number of positions for which the formula body yields "true" provided that for all other positions the body yields "false"; this number is of the predefined type `number`. If there is some position for which the formula body yields "unknown", also the result of the term is "unknown".

In the case of `value`, ⟨*ID*⟩ must denote a binary value function f and ⟨*term1*⟩ and ⟨*term2*⟩ must denote values. The function's first parameter must have the same type as ⟨*term1*⟩ (which must be also the result type of the function); its second parameter must have the same type as ⟨*term2*⟩. The term then denotes the value that results from the combination of the value v_0 of ⟨*term1*⟩ with all values resulting from the evaluations of ⟨*term2*⟩ (in the following description, we assume that these are the values v_1, v_2, v_3 listed in the increasing order of the corresponding positions).

The mandatory evaluation specifier ⟨*mode2*⟩ describes how the combination proceeds. It can have one of the following values:

- `par`: in this case, the two parameters of function f must have the same type. The order in which the function is applied to the individual values v_0, v_1, v_2, v_3 is completely undefined, it could e.g. be

  $$f(f(f(v_3, v_1), v_0), v_2)))$$

  The result is therefore only uniquely defined if f is both commutative and associative.

- `seq`: in this case, the two parameters of function f may have different types. The function is applied in the increasing order of positions yielding the result

  $$f(f(f(v_0, v_1), v_2), v_3)))$$

- `strict`: in this case, the function is applied in the same way as in `seq`. Additionally however, during the evaluation the constants `old` and `new` are defined for access to the current state of the combination in `until` and `while` formulas (see Sections G.1 and F.4).

Pragmatics For the `num` term, the evaluation of all instances proceeds in parallel. The type `number` is externally mapped to the .NET type `System.UInt64`.

Also for the `value` term in the evaluation modes `par` and `seq`, the evaluation of ⟨*term1*⟩ and of all instances of ⟨*term2*⟩ proceed in parallel. However, in mode `par` the combination function is applied whenever a new result becomes available, while in mode `seq` the intermediate results are buffered until the combinations can be performed in the desired order.

In evaluation mode `strict`, first ⟨*term1*⟩ is evaluated. The evaluation of a new instance of ⟨*term2*⟩ only starts after the evaluation of the previous instance (respectively of the initial value ⟨*term1*⟩) has been completed and its value has been integrated into the intermediate result.

Example The term

```
value[par,Zero(),Sum]<IP> x with y <= _ <=# x + 1000 : @x
```

combines the values `@x` for all positions `x` in the denoted range in an arbitrary order with the initial value `Zero()` by application of the function `Sum`.

G.5 Quantified Stream Terms

Grammar

`stream` [[⟨*mode*⟩]] ⟨*variable*⟩ ⟨*term*⟩
`stream` [⟨*mode2*⟩ , ⟨*term1*⟩ , ⟨*ID*⟩] ⟨*variable*⟩ ⟨*term2*⟩
`merge` [[⟨*mode*⟩]] ⟨*variable*⟩ ⟨*term*⟩

Description In all terms, the body term (⟨*term*⟩ respectively ⟨*term2*⟩) is evaluated in all contexts where the local position variable introduced by ⟨*variable*⟩ is bound to all positions denoted by ⟨*variable*⟩ (see Section L).

- In the first `stream` term, ⟨*term*⟩ must denote a value. The result is the stream of all values of ⟨*term*⟩ for the denoted evaluations.

  If ⟨*mode*⟩ is not given or `seq`, the values are ordered in the increasing order of the corresponding positions; if ⟨*mode*⟩ is `par`, the order of the values is undefined (more precisely, the values are placed on the result stream whenever they become available).

- Also in the second `stream` term, ⟨*term*⟩ must denote a value. This construct behaves exactly like the `value` term described in Section G.4, except that its result is the stream of all intermediate combination values. For instance, if ⟨*mode2*⟩ is `seq`, the stream consists of the values

  v_0
  $f(v_0, v_1)$
  $f(f(v_0, v_1), v_2)$
  $f(f(f(v_0, v_1), v_2), v_3)$
  ...

- In the `merge` term, ⟨*term*⟩ must denote a stream. The result is the stream that is constructed by merging the values of all the streams resulting from the denoted evaluations of ⟨*term*⟩.

  If ⟨*mode*⟩ is `seq`, the streams are sequentially concatenated (if some stream is infinite, no message from a subsequent stream will appear in the result stream); if ⟨*mode*⟩ is not given `par`, the order of the messages is undefined (more precisely, the values are placed on the result stream whenever they become available, i.e., the streams are then indeed "merged").

In all cases, every messages in the result stream receives as its time the time of the execution step, when the message has been delivered to the result stream.

Pragmatics It should be noted that the result streams can be finite or infinite.

The default ⟨*mode*⟩ for `stream` is `seq`, because the usual assumption is that stream values are ordered according to the positions to which the quantified variable is bound; the default ⟨*mode*⟩ for `merge` is `par`, because the term "merging" hints towards the non-deterministic combination of stream values.

Example The term

```
stream[seq,Zero(),Sum]<IP> x with y <= _ <=# x + 1000 : @x
```

creates the stream of all partial sums of the values in stream `IP` in the denoted range.

G.6 Terms with Local Bindings

Grammar

⟨*binder*⟩ `:` ⟨*term*⟩

Description The body ⟨*term*⟩ is evaluated in a context where a new variable has been introduced and its value been defined by ⟨*binder*⟩ (see Section J).

Pragmatics The evaluation of the defining phrase in ⟨*binder*⟩ and of ⟨*term*⟩ proceed simultaneously; in order words, it is not the case that the variable introduced in ⟨*binder*⟩ must have its ultimate value before the evaluation of ⟨*term*⟩ can start.

Example The term

```
value<int> y = F(@x) : G(y)
```

introduces a local value `y` defined as `F(@x)` for the evaluation of `G(y)`.

H Monitors

Grammar

⟨*monitor*⟩ ::=
    ⟨*formula*⟩
  | `monitor` ⟨*variable*⟩ ⟨*monitor*⟩

Description A monitor denotes a set of position vectors each of which describes a combination of positions in the monitored streams that violate a desired property. The elements of the set emerge as soon as sufficiently many stream positions have been observed to be able to decide that the property has been violated. Every element is either tagged as a true "violation" (if the monitored property has indeed value "false") or just as a "warning" (if the monitored property has value "unknown", i.e., if it has encountered some undefined term on which the value of the property depends).

If the monitor is a plain ⟨*formula*⟩, the set is either empty or consists of a single position vector of length 0. The ⟨*formula*⟩ then expresses a basic proposition (which may due to free or locally introduced position variables also involve streams). If this property is false, the monitor returns a position vector of length 0.

If the monitor has form `monitor` ⟨*variable*⟩ ⟨*monitor*⟩, then the ⟨*monitor*⟩ may have free occurrences of the position variable introduced by ⟨*variable*⟩ (see Section L). If the basic ⟨*monitor*⟩ returns for some position p of this variable a position vector of some length n, p is prepended to this vector yielding a position vector of length $n + 1$.

The "type" of a monitor is the sequence of stream identifiers denoted in the ⟨*variable*⟩ sections of the nested `monitor` clauses of which the monitor is composed. This sequence must correspond to the sequence of stream identifiers of the monitor declaration (see Section E.6) in which the monitor appears:

`monitor<`S_1`,`S_2`,...,`S_n`>` M =
    `monitor<`S_1`>` x_1 ... `monitor<`S_2`>` x_2 ... `monitor<`S_n`>` x_n `:` ⟨*formula*⟩

Pragmatics Logically, a monitor is similar to a universally quantified formula, i.e., `monitor` ⟨*variable*⟩ ⟨*monitor*⟩ is similar to `forall` ⟨*variable*⟩ ⟨*monitor*⟩. However, the later terminates its evaluation with result "false" as soon as a violating position has been detected. On the contrary, the former continues its evaluation to detect and report all violating positions.

Example The monitor defined as

```
monitor<> M1 = forall<S> x : P(@x);
```

is a plain formula that is true if stream S holds only values with property P. If some value does not satisfy this property, the formula is false, and the monitor terminates with an empty position vector as its result.

The monitor defined as

```
monitor<S> M2 = monitor<S> x : P(@x)
```

reports all positions p of stream S that do not satisfy property P by vectors of form ⟨S = p⟩.

The monitor defined as

```
monitor<S,T> M3 = monitor<S> x: monitor<T> y with x <# _: P(@x,@y)
```

monitors two streams S and T for combinations of positions p in S and q in T such that q occurs in time later than p and the stream values at those positions violate property P. Violating combinations of such positions are reported as vectors of form ⟨S = p, T = q⟩.

I Evaluation Modes

Grammar

⟨*mode*⟩ ::= `seq` | `par`

⟨*mode2*⟩ ::= `seq` | `par` | `strict`

Description Various formulas (see Section F) and terms (see Section G) may contain evaluation mode indicators. The exact meaning of the indicator is indicated in the description of the corresponding phrase. Generally speaking:

- Mode `seq` indicates that the various parts of the phrase are to be evaluated "sequentially", i.e., the evaluation of a later part only begins when the evaluations of the previous parts have been completed.
- Mode `par` indicates that the various parts of a phrase are to be evaluated "in parallel", i.e., the evaluations of all parts begin as soon as the evaluation of the whole phrase begins.
- Mode `strict` in combining `value` terms (see Section G.4) and stream terms (see Section G.5) starts the evaluation of one instance of the body term only after the evaluation of the previous instance has terminated; this allows the use of constants `old` and `new` in the `until` respectively `while` clause of the quantified variable (see Section L).

Pragmatics Parallel evaluation minimizes the delays in the evaluation of phrases and correspondingly the requirements to preserve the histories of streams in memory. If the semantics of the phrase allows parallel evaluation, mode `par` should be thus generally preferred. An exception may be the evaluation of conditional formulas (see Section F.3) and conditional terms (see Section G.2) where the evaluation of both branches may be wasteful.

J Binders

Grammar

⟨*binder*⟩ ::=
    `logical` ⟨*ID*⟩ = ⟨*formula*⟩
  | `position` < ⟨*sid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩
  | `value` < ⟨*tid*⟩ > ⟨*ID*⟩ = ⟨*term*⟩

Description A binder introduces a locally defined variable ⟨*ID*⟩ in a phrase and assigns it its value. There are three kinds of binders:

1. A `logical` binder introduces a logical variable whose value is defined by a ⟨*formula*⟩.
2. A `position` binder introduces a position variable for a stream ⟨*sid*⟩ whose value is defined by a ⟨*term*⟩ (which must denote a position in stream ⟨*sid*⟩).
3. A `value` binder introduces a value variable of some type ⟨*tid*⟩ whose value is defined by a ⟨*term*⟩ (which must denote a value of type ⟨*tid*⟩).

An inner binding may override the definition of a variable given by an outer binding.

Pragmatics The value of a variable is shared by all subsequent occurrences of the variable, i.e., multiple occurrences of a variable do not cause multiple evaluations of the defining phrase.

Furthermore, the introduction of a variable by a binding does not block the evaluation of the phrase in the context of the binding; this evaluation starts even if the value of the variable has not been defined yet and only blocks when this value is required.

The current implementation assumes that every stream is uniquely identified by a global identifier; for this reason, binders for stream variables are not supported.

Example The bindings in the phrase

```
value<int> v = F(@x) : logical p = P(v) : p && Q(v)
```

introduce a local value variable `v` and a logical variable `p` in the evaluation of `p && Q(F(v))`. The value of that phrase is the same as that of the phrase

```
P(F(@x)) && Q(F(@x))
```

However, while in the later phrase `F(@x)` is evaluated twice, it is only evaluated once in the former one.

K Parameters

Grammar

⟨*parameter*⟩ ::=
    `position` < ⟨*sid*⟩ > ⟨*ID*⟩
  | `value` < ⟨*tid*⟩ > ⟨*ID*⟩
  | `stream` < ⟨*tid*⟩ > ⟨*ID*⟩

Description A parameter describes the kind of argument that a function (see Section E) accepts in the position of the parameter. There are three kinds of parameters:

1. A `position` parameter indicates that the argument must be a position for the denoted stream ⟨*sid*⟩.
2. A `value` parameter indicates that the argument must be a value of the denoted type ⟨*tid*⟩.
3. A `stream` parameter indicates that the argument must be a stream whose values have the denoted type ⟨*tid*⟩.

Position parameters that follow a stream parameter may refer to the identifier ⟨*ID*⟩ of the stream parameter as their stream identifier ⟨*sid*⟩. The function may thus be applied to positions from various streams.

Pragmatics In internally defined functions, arguments are inlined for the formal parameters.
In externally defined functions, parameters are handled as follows:

1. For a `position` parameter, the application of the function is delayed until the position becomes available. If the position is undefined, the result of the function application is undefined. Otherwise, the position is passed to the external function as a value of the .NET type `System.UInt64`.
2. For a `value` parameter, the application of the function is delayed until the value becomes available. If the value is undefined, the result of the function application is undefined. Otherwise, the value is passed to the external function as a value of the .NET type `System.Object`.
3. For a `stream` variable, the function can be invoked without delay with a value of the .NET type `System.Object[]` that contains the values that have appeared on the stream since the last application of the function.

   If the result of the function is not a `stream`, the function is thus only invoked once with values that are available on the stream at the time of the function invocation.

   However, if the result of the function is a `stream`, the function is invoked at every execution step with the values that are available on the stream since the last application (see Section E.5).

   Thus, if a function has a `stream` parameter, it should also have a `stream` result to be pragmatically useful.

Example The function declaration

```
stream<int>
mergeAdd(stream<int> S1, stream<int> S2, value<int> v);
```

describes the interface of an externally defined function that merges the values of two streams into a result stream and adds a value to each result. The corresponding externally defined C# function may have the following interface:

```
public static System.Object[]
mergeAdd(System.Object[] S1, System.Object[] S2, System.Int64 v)
```

L Quantified Variables

Grammar

⟨*variable*⟩ ::= < ⟨*sid*⟩ > ⟨*ID*⟩
[`with` ⟨*bound*⟩ { `and` ⟨*bound*⟩ }]
{ ⟨*constraint*⟩ }
[(`until` | `while`) ⟨*formula*⟩) `:`

⟨*bound*⟩ ::=
⟨*boundvalue*⟩ ⟨*relation*⟩ _
| _ ⟨*relation*⟩ ⟨*boundvalue*⟩
| ⟨*boundvalue*⟩ ⟨*relation*⟩ _ ⟨*relation*⟩ ⟨*boundvalue*⟩

⟨*boundvalue*⟩ ::= ⟨*term*⟩ [(+ | -) ⟨*TIME*⟩]

⟨*relation*⟩ ::= < | <= | <# | <=#

⟨*constraint*⟩ ::=
`satisfying` ⟨*formula*⟩
| ⟨*binder*⟩

Description A ⟨*variable*⟩ declaration may quantify certain formulas (see Section F), terms (see Section G), and monitors (see Section H) with a position variable ⟨*ID*⟩ that runs over a certain (finite or infinite) range of positions of the denoted stream ⟨*sid*⟩.

The evaluation of such quantified phrases proceeds by creating a sequence of bindings of variable ⟨*ID*⟩ to all positions of the denoted range, from the lowest position in the range to the highest one (in that order). For the generated binding, the various instances of the body of the phrase are created and evaluated as required by the semantics of the phrase. The creation of new bindings proceeds simultaneously with the evaluation of the instances: whenever a new binding can be created, it is used for the creation of a new instance; this instance is added to the pool of previously created instances and from now on takes part in their subsequent evaluations.

The evaluation of quantified phrases is subject to the following fundamental constraint:

A binding of a position variable to a stream position (and the corresponding instance) is created only after that position has been observed in the stream, i.e., when the stream has received a value at that position.

This constraint ensures that in an instance of the body of the quantified phrase the value at the denoted stream position is already available such that the evaluation of the phrase is not blocked by looking up this value; it also ensures that instances are created "in pace" with the elaboration of the stream, i.e., that the system is not overloaded by a suddenly created large number of instances that refer to the future of the stream (and probably will be immediately blocked by looking up stream values at positions that are not yet available).

The range of the quantified positions may be constrained by an list of bound declarations that are introduced by the keyword `with` and separated by `and`. Each such declaration has the form

⟨*boundvalue*⟩ ⟨*relation*⟩ _ ⟨*relation*⟩ ⟨*boundvalue*⟩

where either part before or after the token _ (which represents the quantified variable itself) can be omitted. Each ⟨*boundvalue*⟩ can be one of the following:

1. A ⟨*term*⟩ that must denote a position: if a position relation (see below) is applied to this position, this position must be on the same stream as the quantified variable; the ⟨*boundvalue*⟩ then denotes this position. If a time relation (see below) is applied to this position, this position may be on any stream; the whole phrase then denotes the time of the value in this stream on this position.

2. A phrase ⟨*term*⟩+⟨*TIME*⟩ or ⟨*term*⟩-⟨*TIME*⟩ where ⟨*term*⟩ must denote a position (on any stream): the ⟨*boundvalue*⟩ then denotes a time, more concretely, the time of the value in that stream on that position plus/minus the offset indicated by the ⟨*TIME*⟩ literal.

Each ⟨*relation*⟩ can be one of the following:

1. A position relation < or <=: the position indicated to the left must be less than respectively less than or equal the position to the right; both positions must be on the same stream.

2. A time relation <# or <=#: the time (of the position) indicated to the left must be less than respectively less than or equal the time (of the position) indicated to the right; both times may be derived from positions on different streams.

For instance, in the following declaration of a quantified position variable `x` on stream `S`

```
<S> x with y <= _ <# y+1000
```

the range of `x` starts at position `y` (which must denote a position on `S`); it ends with the last position whose time is not more than 1000 time units greater than the time of the value at position `y` in stream `S`.

If there is more than one ⟨*bound*⟩, the constraints of the various bounds are combined; above example is thus equivalent to the following variable declarations:

```
<S> x with y <= _ and _ <# y+1000
<S> x with _ <# y+1000 and y <= _
```

If a variable declaration is constrained by one more more ⟨*bound*⟩ declarations, the evaluation of the quantified phrase is subject to the following constraint:

> *The creation of bindings for a position variable (and thus the evaluation of the quantified phrase) does not start before all positions that are directly referenced in the ⟨bound⟩ phrases have been observed on their corresponding streams.*

Thus in above examples the evaluation of the quantified phrase can only begin when the position `y` has been reached in stream `S`; this is the only constraint, since only `y` is directly referenced in the bound. In particular, it is *not* necessary that the last position whose time is not more than 1000 time units greater than the time at position `y` is reached (because not this position is directly referenced in the bound but only its time; this time can be directly calculated from the time of the value at position `y`).

When a binding for a variable has been generated, it may be transformed by a sequence of ⟨*constraint*⟩ clauses of one of the following forms:

- `satisfying` ⟨*formula*⟩: this clause acts as a "filter"; if for the generated binding the ⟨*formula*⟩ evaluates to "true", the binding is preserved; if the result is "false", the binding is discarded; if the result is "unknown", the position becomes "unknown" (which may or may not affect the evaluation of the quantified phrase).
- ⟨*binder*⟩: this clause extends/overrides the generated binding by the binding for another local variable (see Section J); this binding may be used in the subsequent clauses of the <*variable*> declaration and also in the body of the quantified phrase.

For example, in the variable declaration

```
<S> x value<int> m = @x satisfying P(m)
```

a local variable `m` is introduced that the denotes the value of stream `S` at the position assigned to the quantified variable `x`; if this value does not satisfy the condition `P(m)`, this position is not considered in the evaluation of the quantified phrase.

The ⟨*variable*⟩ declaration may be terminated by a clause of one of the following forms:

- `until` ⟨*formula*⟩: if the ⟨*formula*⟩ is satisfied for the current position p of the quantified variable, then p is the maximum of the quantification range (p itself is included in the range); no more binding for a position greater than p will be generated
- `while` ⟨*formula*⟩: if the ⟨*formula*⟩ is *not* satisfied for the current position p of the quantified variable, then p is not part of the quantification range any more; furthermore, no more binding for a position greater than p will be generated.

For example, let us assume that 3 is smallest position at which stream `S` holds a value that satisfies predicate `P`. Then the variable declaration

```
<S> x until P(@x)
```

denotes the positions $0, 1, 2, 3$ while the variable declaration

```
<S> x while !P(@x)
```

denotes the positions $0, 1, 2$.

Pragmatics The unit of time for a ⟨*TIME*⟩ value is 1 tick of the .NET framework, which corresponds to 100 ns. Therefore 1 ms consists of 10 thousand time units and 1 s consists of 10 million time units.

The concept of `until` respectively `while` clauses was introduced in order to replace frequently required specification patterns such as

position<S> `q = min`<S> `q with p` <= `_ : P(@x)`
⟨*quantifier*⟩ <S> `x with p` <= `x` <= `q : Q(@x)`

by the more efficient form

⟨*quantifier*⟩ <S> `x with p` <= `x until P(@x) : Q(@x)`

In the first pattern, the evaluation of the ⟨*quantifier*⟩ phrase blocks until the upper bound `q` of the quantification range becomes known; in order to evaluate the body `Q(@x)` of the phrase, therefore all stream values of the quantification range have to be buffered. In the second form, no buffering is necessary: each instance of `Q(@x)` can be evaluated as soon as the corresponding binding for the quantified variable `x` has been generated.

M ANTLR 4 Grammar

In the following, we list the grammar used by the parser generator ANTLR 4 [4] to generate the lexical and syntactic analyzer for the specification language.

```
// --------------------------------------------------------------------------
// LogicGuard.g4
// LogicGuard ANTLR 4 Grammar
//
// Copyright (C) 2011- Project LogicGuard (RISC, RISC Software, SecureGUARD).
// All Rights Reserved. See http:///www.risc.jku.at/projects/LogicGuard/
// --------------------------------------------------------------------------

grammar LogicGuard;
options
{
  language=CSharp3;
}

// --------------------------------------------------------------------------
// specifications
// --------------------------------------------------------------------------

specification: (decls+=declaration EOS)* EOF ;

declaration:
  # EmptyDeclaration
| 'type' typeid
  # TypeDeclaration
| 'stream' '<' typeid '>' streamid
  # StreamDeclaration
```

```
| 'stream' '<' typeid '>' streamid '(' (params+=parameter (',' params+=parameter)* )? ')'
  # StreamFunctionDeclaration
| 'stream' '<' typeid '>' streamid '(' (params+=parameter (',' params+=parameter)* )? ')' '=' term
  # StreamFunctionDefinition
| 'stream' '<' typeid '>' streamid '=' term
  # StreamDefinition
| 'logical' logicalid '(' (params+=parameter (',' params+=parameter)* )? ')'
  # LogicalFunctionDeclaration
| 'logical' logicalid '(' (params+=parameter (',' params+=parameter)* )? ')' '=' formula
  # LogicalFunctionDefinition
| 'logical' logicalid '=' formula
  # LogicalDefinition
| 'value' '<' typeid '>' valueid '(' (params+=parameter (',' params+=parameter)* )? ')'
  # ValueFunctionDeclaration
| 'value' '<' typeid '>' valueid '(' (params+=parameter (',' params+=parameter)* )? ')' '=' term
  # ValueFunctionDefinition
| 'value' '<' typeid '>' valueid '=' term
  # ValueDefinition
| 'position' '<' streamid '>' positionid '(' (params+=parameter (',' params+=parameter)* )? ')' '=' term
  # PositionFunctionDefinition
| 'position' '<' streamid '>' positionid '=' term
  # PositionDefinition
| 'monitor' '<' (sids+=streamid (',' sids+=streamid)* )? '>' monitorid '=' monitor
  # MonitorDefinition
;

monitor:
  formula                    # FormulaMonitor
| 'monitor' variable monitor # QuantifiedMonitor
;

// ------------------------------------------------------------------------
// formulas
// ------------------------------------------------------------------------

formula:
  '(' formula ')'                          # ParenthesizedFormula
| 'true'                                   # TrueFormula
| 'false'                                  # FalseFormula
| 'logical' '?'                            # UnknownFormula
| 'defined' formula                        # DefinedFormulaFormula
| 'defined' term                           # DefinedTermFormula
| logicalid                                # VariableFormula
| logicalid '(' (terms+=term (',' terms+=term)* )? ')' # ApplicationFormula
| '!' formula                              # NotFormula
| formula '&&' formula                     # AndFormula
| formula '&&' '[' seq ']' formula         # AndFormulaSeq
| formula '||' formula                     # OrFormula
| formula '||' '[' seq ']' formula         # OrFormulaSeq
| formula '=>' formula                     # ImpliesFormula
| formula '=>' '[' seq ']' formula         # ImpliesFormulaSeq
| formula '<=>' formula                    # EquivalentFormula
| formula '<=>' '[' seq ']' formula        # EquivalentFormulaSeq
| 'if' formula 'then' formula 'else' formula # ConditionalFormula
```

```
| 'if' '[' seq ']' formula 'then' formula 'else' formula # ConditionalFormulaSeq
| 'forall' variable formula                 # ForallFormula
| 'exists' variable formula                 # ExistsFormula
| binder ':' formula                        # BinderFormula
;

// ------------------------------------------------------------------------
// terms
// ------------------------------------------------------------------------

term: termcore ( '[' termannotation ']' )? ;

termannotation:
  'position' '<' streamid '>' # PositionTermType
| 'value' '<' typeid '>'      # ValueTermType
| 'stream' '<' typeid '>'     # StreamTermType
;

termcore:
  '(' term ')'                                          # ParenthesizedTerm
| 'value' '<' typeid '>' '?'                            # UnknownValueTerm
| 'stream' '<' typeid '>' '?'                           # UnknownStreamTerm
| 'stream' '<' typeid '>' 'empty'                       # EmptyStreamTerm
| 'position' '<' streamid '>' '?'                       # UnknownPositionTerm
| 'zero' '<' streamid '>'                               # ZeroPositionTerm
| 'old'                                                 # OldValueTerm
| 'new'                                                 # NewValueTerm
| ident                                                 # VariableTerm
| ident '(' (terms+=term (',' terms+=term)* )? ')'      # ApplicationTerm
| streamid '@' term                                     # ContentValueTerm
| '@' term                                              # ContentValueTermCore
| streamid '#' term                                     # TimeValueTerm
| '#' term                                              # TimeValueTermCore
| 'if' formula 'then' term 'else' term                  # ConditionalTerm
| 'if' '[' seq ']' formula 'then' term 'else' term      # ConditionalTermSeq
| 'min' variable formula                                # MinPositionTerm
| 'max' variable formula                                # MaxPositionTerm
| 'num' variable formula                                # NumberValueTerm
| 'value' '[' seq2 ',' term ',' valueid ']' variable term  # CombineValueTerm
| 'stream' '[' seq ']' variable term                    # ConstructStreamTerm
| 'stream' '[' seq2 ',' term ',' valueid ']' variable term # CombineStreamTerm
| 'merge' '['seq ']' variable term                      # MergeStreamTerm
| binder ':' term                                       # BinderTerm
;

seq:
  'seq' # SeqTag
| 'par' # ParTag
;

seq2:
  'seq'    # Seq2Tag
| 'par'    # Par2Tag
| 'strict' # Strict2Tag
```

```
;

// ----------------------------------------------------------------------------
// auxiliaries
// ----------------------------------------------------------------------------

binder:
  'logical' logicalid '=' formula                  # LogicalBinder
| 'position' '<' streamid '>' positionid '=' term  # PositionBinder
| 'value' '<' typeid '>' valueid '=' term          # ValueBinder
;

parameter:
  'position' '<' streamid '>' positionid  # PositionParameter
| 'value' '<' typeid '>' valueid          # ValueParameter
| 'stream' '<' typeid '>' streamid        # StreamParameter
;

variable:
  '<' streamid '>' positionid
  ( 'with' bounds+=bound ( 'and' bounds+=bound )* )?
  ( constraints+=constraint )*
  ( until formula )?
  ':'
;

until:
  'until' # UntilTag
| 'while' # WhileTag
;

bound:
  boundvalue relation '_'                      # LeftBound
| '_' relation boundvalue                      # RightBound
| boundvalue relation '_' relation boundvalue  # LeftRightBound
;

relation:
  '<'   # LessPosition
| '<='  # LessEqualPosition
| '<#'  # LessTime
| '<=#' # LessEqualTime
;

boundvalue:
  term          # TermBoundValue
| term '+' TIME # PlusBoundValue
| term '-' TIME # MinusBoundValue
;

constraint:
  'satisfying' formula # FormulaConstraint
| binder               # BinderConstraint
;
```

```
typeid: ident;
streamid: ident;
logicalid: ident;
positionid: ident;
valueid: ident;
monitorid: ident;

ident: IDENT ;

// ------------------------------------------------------------------------
// lexical rules
// ------------------------------------------------------------------------

// reserve leading underscore for internal identifiers
IDENT : [a-zA-Z][a-zA-Z_0-9]* ;
TIME  : [0-9]+ ;
EOS   :  ';' ;

WHITESPACE  : [ \t\r\n\f]+ -> skip ;
LINECOMMENT : '//' .*? '\r'? ('\n' | EOF) -> skip ;
COMMENT     : '/*' .*? '*/' -> skip ;

// matches any other character
ERROR : . ;

// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```